2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

Real-time Scheduling and Synchronization for the NPS Autonomous Underwater Vehicle

by

Dionysios Makris

December, 1991

Thesis Advisor:	Shridhar B. Shukla
Second Reader	Roberto Cristi

Unclassified
Security Classification of this page

REPORT DOCUMENTATION PAGE

1a. Report Security Classification: UNCLASSIFIED	1b. Restrictive Markings
2a. Security Classification Authority	3. Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b. Declassification/Downgrading Schedule	
4. Performing Organization Report Number(s)	5. Monitoring Organization Report Number(s)
6a. Name of Performing Organization: Naval Postgraduate School — 6b. Office Symbol *(if applicable)*: Code 33	7a. Name of Monitoring Organization: Naval Postgraduate School
6c. Address *(City, State, and ZIP Code)*: Monterey, CA 93943-5000	7b. Address *(City, State, and ZIP Code)*: Monterey, CA 93943-5000
8a. Name of Funding/Sponsoring Organization — 8b. Office Symbol *(if applicable)*	9. Procurement Instrument Identification Number
8c. Address *(City, State, and ZIP Code)*	10. Source of Funding Numbers: Program Element Number — Project No. — Task No. — Work Unit Accession No.

11. Title *(Include Security Classification)* REAL-TIME SCHEDULING AND SYCHRONIZATION FOR THE NPS AUTONOMOUS UNDERWATER VEHICLE

12. Personal Author(s) MAKRIS, Dionysios

13a. Type of Report	13b. Time Covered	14. Date of Report *(Year, Month, Day)*	15. Page Count
Master's Thesis	From ______ To ______.	December 1991	97

16. Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. Cosati Codes

Field	Group	Subgroup

18. Subject Terms *(Continue on reverse if necessary and identify by block number)* Aperiodic, Data-flow, Periodic, Rate Monotonic, Real-time, Scheduling, Synchronization, Task.

19. Abstract *(Continue on reverse if necessary and identify by block number)*

The work described in this thesis is part of a multi-year research project to develop an Autonomous Underwater Vehicle (AUV-II), which is an intelligent robot submarine, carried out by the departments of Mechanical Engineering, Computer Science and Electrical and Computer Engineering of the Naval Postgraduate School.

The AUV-II on-board computer must perform several different tasks such as navigation, autopilot, guidance, sonar processing, and collision avoidance, etc. under strict timing constraints to guarantee the safety of the vehicle. This thesis describes the design and development of real-time scheduling software, which is capable of scheduling and synchronizing the periodic and aperiodic processes required by the AUV-II. A design recommendation of a Graphical User Interface has been developed to improve the software engineering aspects of this project.

20. Distribution/Availability of Abstract: [✓ unclassified/unlimited] ☐ same as report ☐ DTIC users	21. Abstract Security Classification: UNCLASSIFIED
22a. Name of Responsible Individual: Shridhar B. Shukla	22b. Telephone *(Include Area Code)*: (408) 646-2764 — 22c. Office Symbol: EC/Sh

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — security classification of this page

Unclassified

Real-time Scheduling and Synchronization for the NPS Autonomous Underwater Vehicle

by

Dionysios Makris
Lieutenant JG, Hellenic Navy
B.S, Hellenic Naval Academy, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

December 1991

Author: [illegible]
Dionysios Makris

Approved by: [illegible]
Shridhar B. Shukla, Thesis Advisor

[illegible]
Roberto Cristi, Second Reader

Michael A. Morgan
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The work described in this thesis is part of a multi-year research project to develop an Autonomous Underwater Vehicle (AUV-II), which is an intelligent robot submarine, carried out by the departments of Mechanical Engineering, Computer Science, and Electrical and Computer Engineering of the Naval Postgraduate School.

The AUV-II on-board computer must perform several different tasks such as navigation, autopilot, guidance, sonar processing, and collision avoidance, etc. under strict timing constraints to guarantee the safety of the vehicle. This thesis describes the design and development of real-time scheduling software, which is capable of scheduling and synchronizing the periodic and aperiodic processes required by the AUV-II. A design recommendation of a Graphical User Interface has been developed to improve the software engineering aspects of this project.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I owe some thanks to a few people who supported me all the way along to make the completion of this thesis possible. I am greatful to my advisor Shridhar B. Shukla. Without his help, wise advice, and cooperation, I wouldn't have finished my thesis.

I would also like to acknowledge my wife, Gelly, for the understanding and love she provided me during this hectic period of our lives.

I. INTRODUCTION

A. BACKGROUND

1. NPS AUV-II

The Naval Postgraduate School is currently developing an Autonomous Underwater Vehicle, AUV-II, which is an intelligent, robot submarine. Figure 1.1, which has been provided by the NPS Computer Science Department, displays an internal layout of the AUV-II. The microprocessor that is used is a Motorola 68030 with 2.5 Mbytes of RAM and 4 Mbytes of EPROM, centered around a twelve slot G-96 bus supplied by the GESPAC corporation. It also has a 200 megabyte hard disk, parallel and serial communication ports, and analog to digital and digital to analog channels.

The operating system is OS-9, developed by Microware Systems Corporation [Ref. GES88]. It is an operating system designed to support real-time applications. It has a built-in editor, file system, compilers, ability to support local area networks, etc. The real-time features that OS-9 supports are timers, process creation and deletion, process priority assignment, process scheduler, synchronization primitives etc.

2. Data-flow for On-board Processing

The AUV-II, in a typical mission, needs eight processes to be running. Six of them are periodic and two aperiodic. These processes must communicate with each other and implement the flow diagram that Fig. 1.2 displays [Ref. Fl91]. The periodic processes are: execute mission, guidance, autopilot, process sonar data, navigation, and monitor system status. They are executed at 10 Hz. Aperiodic

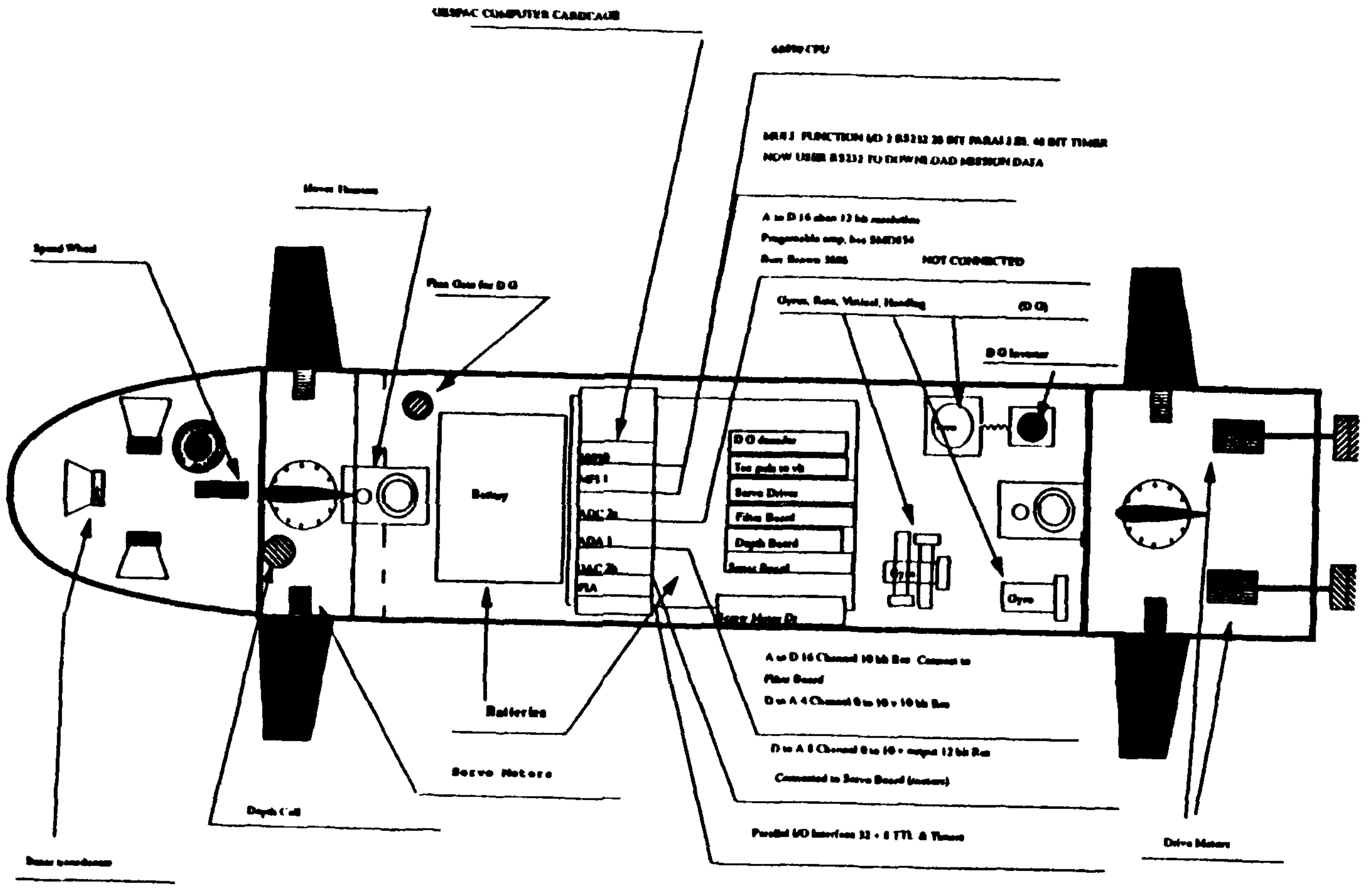

UNIPAC COMPUTER CARDCAGE
68000 CPU
MRU FUNCTION I/O 2 RS232 20 BIT PARALLEL 40 BIT TIMER
NOW USES RS232 TO DOWNLOAD MISSION DATA
A to D 16 chan 12 bit resolution
Programable amp. Ser 5MD814
Burr Brown 3606
NOT CONNECTED
Gyros, Rate, Vertical, Heading
(D G)
D G Inverter
Speed Wheel
Hover Thrusters
Plan Gear for D G
Battery
Batteries
Servo Motors
Depth Cell
Sonar transducers
68020
MPI 1
ADC 2b
ADA 1
DAC 2b
PIA
D G decoder
Test pads to vb
Servo Driver
Filter Board
Depth Board
Sonar Board
Servo Motor Dr
Gyro
Gyro
A to D 16 Channel 10 bit Res Connect to
Filter Board
D to A 4 Channel 0 to 10 v 10 bit Res
D to A 8 Channel 0 to 10 v output 12 bit Res
Connected to Servo Board (motors)
Parallel I/O Interface 32 + 8 TTL & Timers
Drive Motors

Figure 1.1: Internal layout of AUV-II.

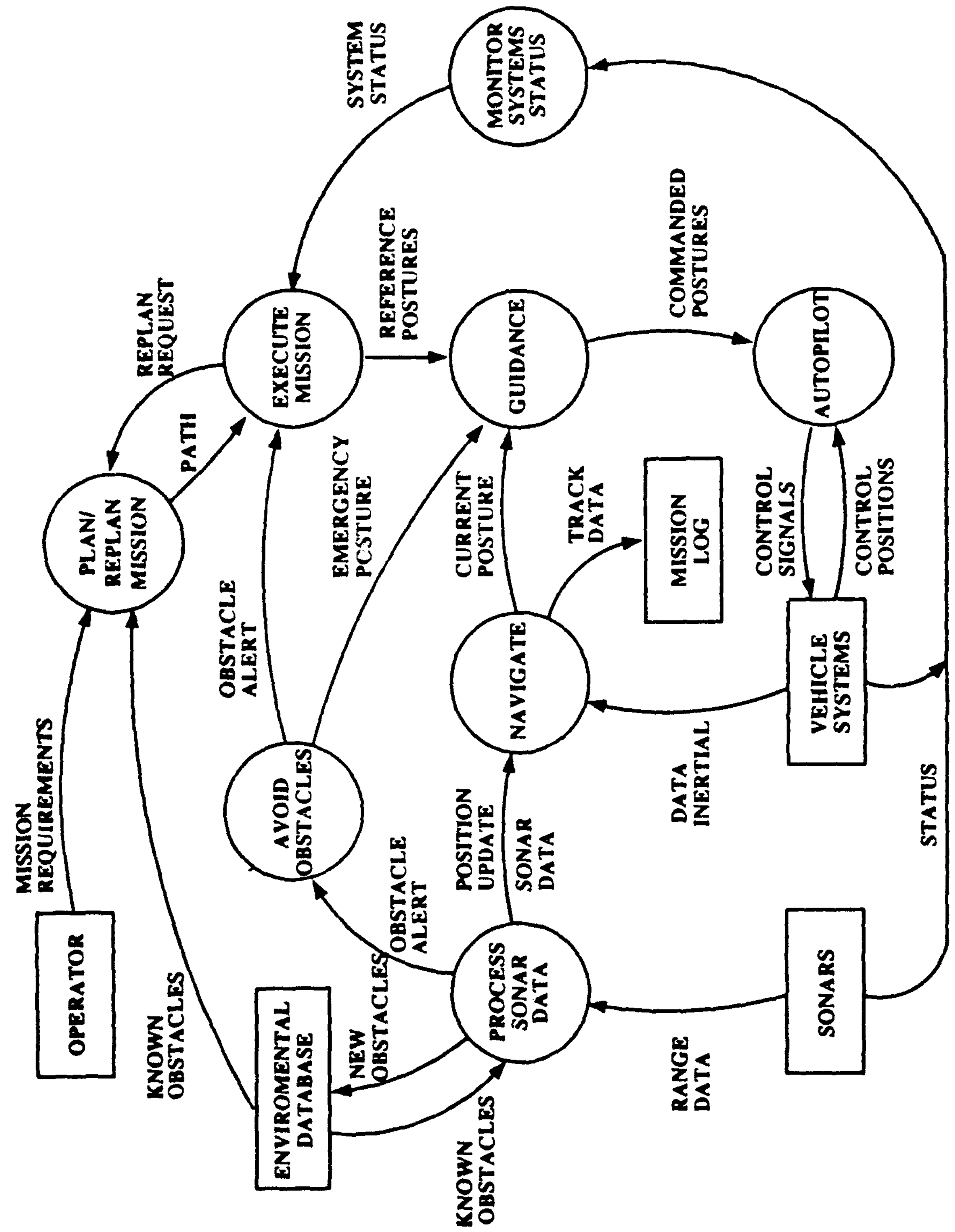


Figure 1.2: Data-flow diagram for on board processing.

processes are: avoid obstacle and plan/replan mission. The AUV-II, by executing these processes in the appropriate order, and by having all the required information from its sensors, completes its mission successfully. The main sensors with which the AUV-II is equipped are gyros for dead reckoning and sonar for real-time navigation and target identification.

3. AUV-II Real-time Programming Environment

At present, the code that is used for the operation of the AUV-II is a single sequential program. This program mainly consists of the initialization part and a loop. In the initialization part, the user provides the operation instructions. In the timed main loop, the program executes all the required instructions for the AUV-II operation. Execution in a single loop is very convenient for applications that execute only a few tasks at the same frequency. It is extremely difficult to design this loop for a large number of tasks, especially when those tasks have different frequencies or some of them have to execute in an aperiodic fashion.

This study separates the operation instructions from the main program to form a set of independent tasks. In this thesis, the words task and process are used interchangeably. The main program consists only of functions that are needed before the start of the execution. The user has only to specify the processes that the AUV-II has to execute and initialize the execution values. Since the user of this program is likely not to be an expert in real-time systems, a user interface capable of hiding all the details from the user is created.

The whole design is based on the currently used hardware platform but effort has been made to make the software compatible with future hardware upgrade. In case this is not possible, the general ideas of the design are selected in such a way that they can be used even under major hardware modifications. As an example, the selection of the OS-9 PIPE as a synchronization primitive that is discussed in

Chapter III, is still valid if the host computer is replaced by a transputer board, and the use of pipes is replaced with the use of messages.

B. OBJECTIVES OF THE STUDY

This thesis is a continuation of [Ref. Le91], in which a scheduler capable of scheduling **independent** periodic processes according to the rate monotonic algorithm was created. The objectives of this thesis are:

- Create a scheduling scheme capable of scheduling periodic and aperiodic processes.
- Provide the required synchronization in processes dependent on each other.
- Study the effect of the synchronization on schedulability under rate monotonic algorithm.
- Verify the capability of the scheduling scheme to be used in the AUV-II.
- Design a framework for a Graphical User Interface (GUI) that will be used when the current host computer is replaced by a portable workstation.

C. THESIS ORGANIZATION

Chapter II presents the scheduling and synchronization requirements of real-time systems. It also presents how the schedulability is affected by the synchronization. Chapter III presents, the implementation of scheduling and synchronization selected for the AUV-II in detail. This chapter also presents the performance of the selected scheduling scheme together with the experiments that are used to verify the performance. Chapter IV presents a design of a Graphical User Interface that can be implemented when the current computer is replaced by a portable workstation. Finally Chapter V concludes what this study has achieved and provides some directions for further improvement.

II. ISSUES IN REAL-TIME SYSTEMS

A computer system is called real-time when it can support the execution of real-time applications. Real-time applications differ from ordinary computer applications because they have strict timing requirements. These requirements almost always are connected with *deadlines* of the tasks, which is the time that a task has to finish its computation. Deadlines can be either *hard* or *soft*. If the results of a computation are useless after the deadline, it is called hard; if their validity only starts to degrade then it is called soft.

In most real-time applications, more than one task has to run at the same time. It is easy to handle that if one processor is assigned for every task. In uniprocessor applications like the AUV-II, this cannot happen. For that purpose, a multitasking operating system that fakes multiple processors, like OS-9 does, has to be selected.

A. REAL-TIME SCHEDULING

Whenever more than one task is runnable, there has to be a mechanism that decides which one to run first. This decision mechanism can be a part of the operating system, it can be a process outside the operating system called as application scheduler, or it can be a part of the running tasks. Its objective is to assist the scheduler in the operating system according to the application requirements. It is one of the most important parts in a real-time application because the decisions of the scheduler will determine how the computation time of the processor is going to be used in the most effective way, and therefore, if any deadlines are going to be missed.

Complete analysis of the different scheduling algorithms can be found in [Ref. Ta87]. As reported therein, a scheduling algorithm, in order to be effective, has to be *priority driven*, and not *priority independent.* This is because all tasks are not equally important at a specific time and because priority independent scheduling cannot provide any guarantees that the timing constraints are satisfied. The algorithm has to be *preemptive*, as against *run to completion*, in order to stop a task that is being executed for a more important one in such a way that as few deadlines as possible are missed.

1. Static Scheduling

A scheduling algorithm is said to be *static* or *fixed* if the priority is assigned once in the process (probably before the beginning of the application) [Ref. LA90]. An example of a static algorithm is the *rate monotonic algorithm* [Ref. LL73]. These algorithms can guarantee only average performance, since the decision of the scheduler is based on the assigned priorities and not on the current conditions. For periodic tasks, the priorities are assigned according to a predetermined characteristic like the period. For aperiodic tasks, the priority is also assigned according to predetermined characteristics, but, depending on whether the deadlines are hard or soft, either the average value or the worst case value can be used.

2. Dynamic Scheduling

A scheduling algorithm is said to be *dynamic* or *time driven* when the priority can change with time. Examples of dynamic algorithms are *earliest deadline*, *minimal laxity*, etc. These algorithms have a better performance than static ones. Most of the dynamic algorithms handle the periodic and aperiodic tasks in the same way, since the decision of the scheduler is based on the current conditions. The main drawback in the dynamic algorithms is that they are difficult to implement.[Ref. LA90]

3. Rate Monotonic Scheduling

As has been shown by Liu and Layland, the optimum fixed priority scheduling algorithm is the one that assigns priorities to a set of independent tasks according to their request rate; the higher the request rate, the higher the priority [Ref. LL73]. This is known as rate monotonic priority assignment.

The rate monotonic algorithm can ensure, for n independent tasks, that all the deadlines will be met if the conditions of the theorem proved by Liu and Layland are met [Ref. LL73]. It states that

the upper bound to processor utilization u for a set of n independent tasks is given by:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \qquad (2.1)$$

where C_i is the execution time and T_i the period of the i process.

It is possible to increase the CPU utilization and still meet all the deadlines. This is specified by the theorem [Ref. LSD89] which states that

a set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines if:

$$\forall i, 1 \leq i \leq n, \min_{(k,l)\epsilon R_i} \sum_{j=1}^{i} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \leq 1 \qquad (2.2)$$

where $R_i = \{(k,l) | 1 \leq k \leq i, l = 1, ..., \lfloor \frac{T_i}{T_k} \rfloor\}$, T_j the period and U_j the process utilization of the i process.

Although the rate monotonic algorithm is not the optimal scheduling algorithm, it is very important for the following reasons:

- The most important task, in an independent set of tasks, will meet its deadlines even under temporary overload.
- It provides fast response time for the aperiodic tasks, while it is able to meet all the deadlines for the periodic tasks. This is of great importance for the AUV-II where some of the tasks are aperiodic.
- The priorities can be modified in such a way that the required synchronization between the tasks can be achieved.
- It can be used to schedule tasks where computation time is known imprecisely, as in many AUV-II algorithms.

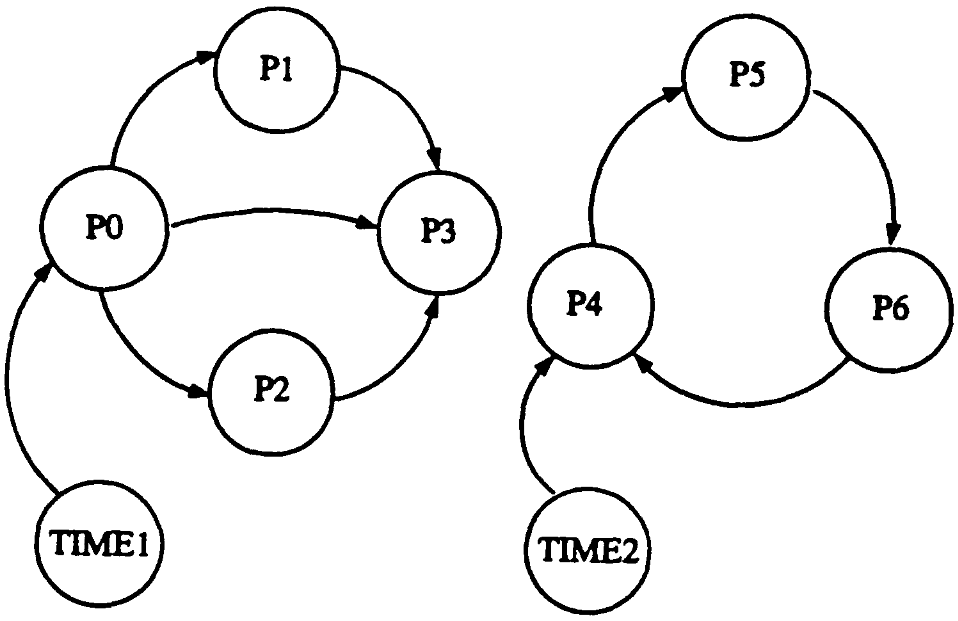


Figure 2.1: Timing mechanism for two independent sets.

4. **Data-flow Scheduling**

Data-flow scheduling is completely different from the previously defined algorithms because it does not need any real-time application scheduler. In this scheme of scheduling, the start of the execution of a process is triggered by receipt of input data. The basic advantage of data-flow scheduling is that it combines synchronization among tasks with scheduling. Other scheduling algorithms, on the other hand, are designed for independent tasks and incorporate additional schemes for synchronization.

In data-flow scheduling, if processes have to execute in a periodic fashion, a timing mechanism has to be inserted for each independent set of processes. This mechanism can be either a delay in one of the processes or an extra process, as in Fig. 2.1, that will implement only the periodicity. This causes all the other processes that depend on this process to also keep the periodicity. Data-flow scheduling can be used even for independent processes, if each process is handled as an independent set. This is not efficient because processor time is spent on each individual timing mechanism and the set of tasks becomes bigger causing slower process switching.

All the previously outlined scheduling algorithms can be used in conjunction with data-flow scheduling in order to make a selection for the next process to run when more than two processes are ready to be executed. This can happen in the case where more than one independent set of processes is being executed, as in Fig. 2.1, or even in the same set when more than two independent processes exist. Data-flow scheduling is suitable for applications like AUV-II where all the processes are closely dependent and only one timing mechanism can achieve the required frequency of execution. The required data consistency is intrinsic to the scheduler.

5. Schedulability Analysis

Schedulability analysis is the process of verifying if a given set of tasks can meet its deadlines when using a specific scheduling algorithm [Ref. TK88]. The use of a schedulability analyzer is very important because it provides a tool to predict if a specific set of tasks can be used before the actual execution.

In order to carry out schedulability analysis, certain basic information about the tasks is required regadless of the scheduling algorithm. This information, for periodic tasks, is the execution time and the period. For aperiodic tasks, it depends on the deadlines. If the deadlines are hard, the execution time, worst case interarrival interval, and expected response time are needed. If the deadlines are soft, the execution time, mean period, standard deviation of the period, and expected response time are needed. In general, this information is called *attributes* of a task. The results required from the schedulability analysis are the CPU utilization, prediction of any possible missed deadlines, tasks which are going to miss a deadline, and expected response time.

B. REAL-TIME SYNCHRONIZATION

The tasks of a real-time application are not usually independent. These dependencies arise from the need for communication between the processes to synchronize with each other for correct and fai sharing of logical or physical resources. Since each task has some limited time bounds within which it has to be executed, these dependencies make the timing requirements more stringent. These requirements result in a reduced utilization of the processor leading to an increased number of missed deadlines for a given processor load. It is important to use the right synchronization protocol in order to provide the highest possible utilization, and at the same time, avoid some undesirable situations like *priority inversion*, where a high priority task is blocked by a lower priority task for an unpredictable period of time [Ref. SRL91].

In most real-time applications, processes which are working together write and read the same shared data. This can cause *race conditions*, in which multiple processes try to use the shared data with unpredictable results. The only way to avoid race conditions is to achieve *mutual exclusion*, that is, when a process is using the shared data, others are excluded from using it. Our objective is to provide synchronization that avoids race conditions and maintains *data consistency.* Data consistency prevents a process from using the shared data before it is updated. There are various primitives which facilitate such synchronization, such as:

Semaphores: A semaphore is a variable with integer values, usually only 0 and 1 [Ref. Di88, Ta87]. Semaphores can be used in such a way that 0 means that the shared data has not be updated or someone else is using the shared data and 1 means that the new data is available and no one else is using it. Since the semaphores are always supported by hardware instructions like TEST AND SET LOCK, mutual exclusion is guaranteed.

Monitors: Monitors are programming language constructs that group together variables, data structures, and procedures [Ref. Ta87]. The most important property of the monitors is that they provide mutual exclusion without any further design by the programmer. When a process is using a monitor, all other processes that compete for access to the monitor are suspended until the monitor is not in use. With proper use of the variables or with one more semaphore, the desired data consistency can also be achieved.

Events: Events are a special kind of variable [Ref. Di88]. The values of events *(E)* change only with instructions supported by hardware, like *Signal(E)* and *Wait(E)*. Proper use of event instructions causes a process to be executed only if the required conditions have been met. As a result, both mutual exclusion and data consistency can be achieved.

Pipes: Pipes are sequential files which never leave the system's RAM memory [Ref. Di88]. Usually these files are small, since data stays there till a process reads from the pipe. What makes pipes useful is that, if a process tries to read from an empty pipe, it is suspended until some other process puts data in the pipe. With the above property, it is ensured that the data is always updated. The main drawback in the use of pipes is that they can be used for communication between only two processes, one to write and the other to read.

The purpose of using priority driven, preemptive scheduling is to break the execution of a lower priority process when a higher priority process is ready to run. If, at the same time, a synchronization primitive like semaphores is used to avoid race conditions, the following problem can occur. Consider two processes, P_1 and P_2 with periods $T_1 > T_2$, that access the same shared data. In rate monotonic assignment, P_1 has a higher priority than P_2. Suppose that P_2 first accesses that data and locks the semaphore that controls the access. If, at that time, P_1 starts execution, it will find the semaphore locked and will have to wait for P_2 to finish its critical section. However, that delay may be longer than the critical section of P_2 itself if P_2 is preempted by a third intermediate priority process. This may lead to uncontrollable blocking.

Such blocking of a higher priority process by a lower priority one, for an unpredictable period of time, is called priority inversion. The easiest solution to this problem is not to allow a process to be preempted at the time it is executing its critical section. However, this solution creates unnecessary blocking of processes not using that shared data and is appropriate only for very short critical sections. In order to avoid priority inversion, better solutions, like the *priority inheritance protocol* [Ref. SRL91], *priority ceiling protocol* [Ref. SRL91] and *stack-based resource allocation* [Ref. Ba90] have been proposed in the literature. Each one of the above has a different way of reducing the timing constraints that created from the synchronization. In the next section, we shall discuss some commonly used synchronization protocols.

C. TECHNIQUES OF REAL-TIME SYNCHRONIZATION

1. Two Phase Locking Protocol

An approach to achieve the required synchronization and data consistency between the tasks is the two-phase locking protocol. This protocol, as implied by its name, accesses the shared data or resources in two phases. In the first phase, a task tries to lock all the synchronization primitives for the data it needs to update or read. If it succeeds in the first phase, the task proceeds to the second phase, where it uses the data and, at the end, releases the locks. If the process is not able to lock all the synchronization primitives, or if some data is not already updated, the process either stops a lower priority process or releases all the locks and starts from the beginning. This technique is not desirable when a high processor utilization is of great importance. Also, the locking of all data structures that are going to be used can cause priority inversion problems.

2. Priority Inheritance and Priority Ceiling Protocol

The complete definition of these two protocols can be found in [Ref. SRL91]. We briefly describe them here. The idea upon which the priority inheritance protocol is based is that when a process with low priority blocks the execution of a higher priority one, it inherits the highest priority of all the processes it blocks for the execution of its critical section. This enables the low priority job to finish its execution without the possibility of being preempted from an intermediate priority job. Under this protocol, although data consistency is achieved, the problems of deadlock and *chained blocking* have not been avoided [Ref. SRL91]. Chained blocking is caused when a process has to wait for more than one lower priority process to unlock the synchronization primitive being used.

In the priority ceiling protocol, the same idea as in priority inheritance is used. However, it guarantees that, if a job is preempted in its critical section, the new job will execute at a priority higher than that of all the preempted critical sections. This is realized by assigning a priority ceiling to each semaphore, which is equal to the highest priority of a task that may use that semaphore. A job is allowed to start the execution of a new critical section only if it has a priority higher than all the priority ceilings for all the semaphores locked by jobs other than itself. In this protocol, unlike the priority inheritance protocol, deadlocks and chained blocking are avoided, but a new kind of blocking, called the ceiling blocking, is present. Also, the priority ceiling protocol creates unnecessary blocking in processes that are never going to use a specific semaphore. This happens because priorities are assigned to semaphores assuming that all processes are dependent on each other.

3. Data-flow Synchronization

In many real-time applications like the AUV-II, where the periods of the processes and the synchronization requirements are known in advance, it is not required to use a complicated protocol, like the priority ceiling protocol, to implement synchronization. This reduces the size of the application scheduler, since there is no need for the extra lists to hold the different priorities of the semaphores, the semaphores that are locked, etc. Also, it reduces the computation time required to select the next process to run, since the scheduler has to select only from those processes that have received the updated data.

In data-flow scheduling, synchronization can be achieved easily using signals between processes. The signals can contain the data to be transferred. A process is ready to start execution only if it has received data from all the preceding processes that have completed execution. A process, when it completes, sends signal to its succeeding processes. Until all such signals are received, a process does not start execution, thus ensuring consistency of data and mutual exclusion.

D. EFFECT OF SYNCHRONIZATION ON SCHEDULABILITY

The application of the two phase locking protocol is limited in real-time applications. It has low processor utilization because a process has to be terminated in the middle of its execution when a resource is not available or when a process with higher priority needs the same resource. Also, under these conditions, the application behavior cannot be predicted correctly since the result is based on the chance that the resources will be available.

For the priority ceiling and the priority inheritance protocols, the following corollary has been proved in [Ref. SRL91].

A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm if the following condition is satisfied:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{\frac{1}{n}} - 1) \qquad (2.3)$$

where u is the utilization factor, C_i is the execution time, T_i the period and B_i is the worst timing blocking of the i process.

Also, Equation (2.2), which enables a larger upper bound of the utilization, gets a new form as below [Ref. SRL91].

A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic for all task phasing if:

$$\forall i, 1 \leq i \leq n, \min_{(k,l)\epsilon R_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k}\right] \leq 1 \qquad (2.4)$$

where $R_i = \{(k,l) | 1 \leq k \leq i, l = 1, \ldots, \lfloor \frac{T_i}{T_k} \rfloor\}$, C_i the execution time, T_i the period, B_i the worst timing blocking of the i process, and U_j the process utilization.

It can be derived from the above two modified theorems that the processor utilization is changed as if one new process is included with period equal to the worst case blocking.

The expected schedulability of the data-flow scheduling combined with the rate monotonic algorithm is almost the same as the one provided for the rate monotonic algorithm. The only difference is that each independent set of tasks is treated as one task. In that case, since the number of tasks, n, in Equations (2.1) and (2.3) is reduced, the upper bound of processor utilization is increased. As an example, consider the seven processes of Fig. 2.1. According to Equation (2.1), the upper bound on processor utilization is expected to be less than or equal to $7(2^{\frac{1}{7}} - 1) = 0.73$. However, since there are only two independent sets of tasks, the processor utilization is expected to be less than or equal to $2(2^{\frac{1}{2}} - 1) = 0.83$. Since, in each independent set, the processes satisfy the condition $\frac{T_m}{T_i} - \lfloor \frac{T_m}{T_i} \rfloor = 0$ (fractional part of $\frac{T_m}{T_i} = 0$) for

$i = 1, 2, \ldots, m - 1$. This means that processes all have the same basic period or an integer multiple of that basic period. According to [Ref. LL73], the utilization bound can further increase to 1.0 when no aperiodic processes are included in the process set.

III. SCHEDULING AND SYNCHRONIZATION FOR AUV-II

At present, the AUV-II scheduler uses the rate monotonic algorithm, appropriately modified to work on the OS-9 operating system. It can schedule only periodic processes and it does not provide any synchronization between the processes. The scheduler first calculates the priorities that will be assigned to the processes according to the rate monotonic algorithm. The priorities are assigned with esufficient spacing so that aging, described later, will not have any effect. The process set is then analyzed for schedulability according to Equation (2.2). If the set is schedulable, the scheduler forks all the processes. The scheduler has an array which keeps the next time at which each process will be ready to start execution. By following these requirements in an infinite loop, the scheduler is able to send wake-up signals to the processes and move them from the sleeping queue to the active queue. The above loop of sending wake-up signals instead of creating and killing the process every time was selected since it does not waste computing time in the process initialization every time it is executed. This real-time scheduling mechanism needs to be augmented to incorporate process synchronization and aperiodic processes. This chapter describes how this is done by using PIPES for data-flow synchronization on OS-9.

A. OS-9 SUPPORT FOR SCHEDULING AND SYNCHRONIZATION

OS-9 is an operating system designed to support real-time applications. It has its own scheduler with two main queues - one with the active processes and the other with the sleeping processes [Ref. Di88]. When a process is forked, it is placed in the active queue according to the assigned priority. The priorities can take values from 1 to 65,556. A process is placed in the sleeping queue if there is a **pause()** command in its source code or if it is waiting for I/O.

The OS-9 scheduler finds out the *current* process at regular intervals. This is the process at the top of the active queue and is the process that runs next. The regular interval is called a *tick* and lasts 10 msec on the system being used.

The main steps in the OS-9 scheduler are the following:

- It checks for sleeping processes that are ready to move in the active queue.
- It checks the current process and the active queue to find out which one has the highest priority to become the next current process.
- It starts the execution of the current process.

One of the most interesting characteristics of the OS-9 scheduler is aging. With aging, priorities of the tasks in the active queue are increased by one at every tick. This characteristic has been included in the OS-9 scheduler to provide fairness, but it causes problems when RMS is attempted on top of OS-9. The priorities of processes must follow the order dictated by RMS, but they must be sufficiently separated to make aging ineffective.

Synchronization can be achieved in OS-9 by using PIPES and Events. There are two kinds of pipes - un-named and named. The main difference between them is that a process cannot write to an unnamed pipe if the reader is not ready to receive the information. Un-named pipes are mainly used by the OS-9 shell. Events and pipes are supported with the C language calls. The most useful ones are described in Fig. 3.1 and Fig. 3.2 respectively [Ref. GES90].

B. IMPLEMENTATION OF DATA-FLOW SYNCHRONIZATION

The new run-time scheduler is highly dependent on the synchronization primitive that has been selected. Both, events and pipes, are primitives which can be used to implement synchronization between the processes of AUV-II. Since events require

Call	Meaning
_ev_creat	Create an event structure
_ev_link	Increment event count
_ev_unlink	Decreament event count
_ev_wait	Process waits until Event in boundaries
_ev_signal	Increment the event variable
_ev_read	Reads the event variable
_ev_set	Sets the event variable

Figure 3.1: Calls associated with events.

Call	Meaning
_GC_Rdy	Tests whether data are ready in buffer
_GC_Size	Returns the size of the pipe buffer
close()	Close an open path
create()	Creates a new pipe
open()	Opens an existing path
read()	Reads bytes from path
readln()	Reads one line from path
write()	Writes bytes to path
writeln()	Writes one line to path

Figure 3.2: Useful calls associated with pipes.

a more complicated program and offer functionality not required in AUV-II, use of pipes was selected. Selection of the appropriate primitive was based on the following considerations:

- The critical section in the communication between the processes must be small and consist only of writing and reading some data.
- Events require construction of the data structure to hold the communication data.
- For each data structure, two events are needed; one implements the mutual exclusion in the use of the data structure and the other guarantees the data consistency.
- The output of one process is used by more than one processes only in one case.

All the above observations strengthen our decision to select pipes as the right synchronization primitive for the AUV-II.

By selecting pipes as the synchronization primitive, some major modification had to be done in the run-time scheduler, developed in [Ref. Le91], that changed completely the way the scheduler worked.

After the communication via pipes was included in the tasks, each task could wait for a signal from the run-time scheduler each time it starts execution and then it could wait for some output to be placed in the pipes that it reads. This could cause a process to stop execution at least twice. Also, as in Fig. 3.3, if the process has not finished the execution at the time that the **wake_up()** signal is sent, the processes misses the next execution.

The solution to the above problem is given in Fig. 3.4. This solution discards the **pause()** at the end of each cycle in the called process and the part of the run-time scheduler that sends the **wake_up()** signal. Thus, the process now waits only for the data to be in the pipe.

The next step creates periodicity in the set of processes. This is done by an extra process created with only the purpose of writing a dummy message on a pipe as

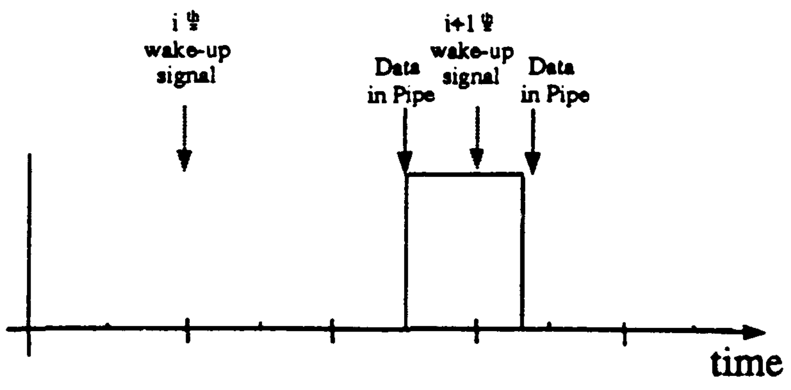


Figure 3.3: Missed execution due to not waiting in a pause.

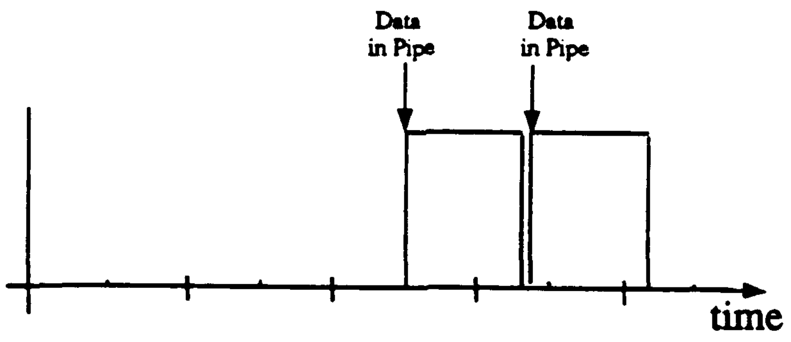


Figure 3.4: Correct number of executions.

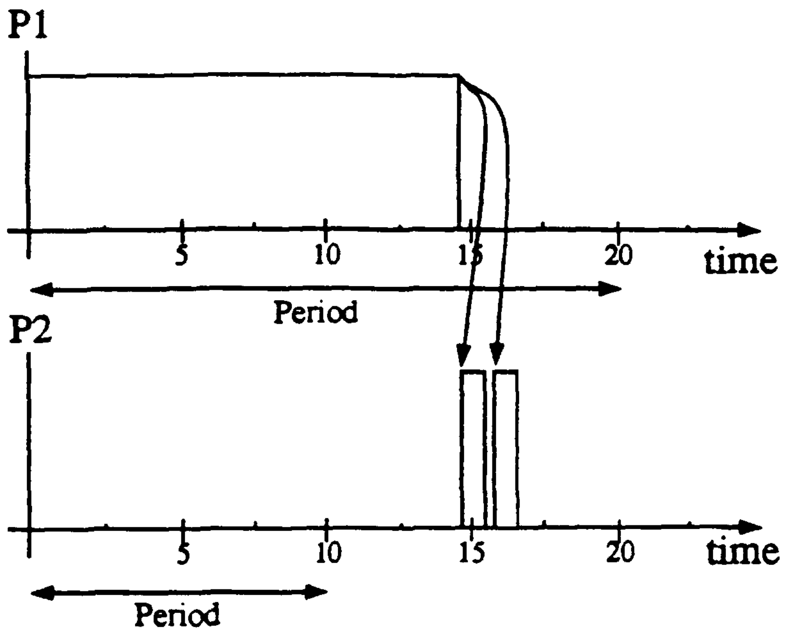


Figure 3.5: No periodic execution in a set of two processes.

in Fig. 2.1. The dummy message is read from the process with the highest frequency in the set. The timing process code is similar to the called processes code, outlined in Appendix C.

This scheme of synchronization has the advantage that, in a temporary processor overload, the dummy messages will be accumulated in the pipe. So, after the overload, the processor will execute all the processes for the required number of times without missing any of them. Thus, the average processor utilization can be close to 1.0.

The above data-flow scheduling cannot guarantee the user that each process will run at exact periodic intervals. This is because of the required communication between processes and it is more likely to happen if a process depends on a process with lower frequency. For example, in Fig. 3.5, the process P1 that has a period of 20 ticks, over an interval of 20 ticks, releases two sets of information. This information is used by process P2 that has a period of 10 ticks over two executions. As a result, P2 is executed twice in the second 10 ticks interval and is not executed in the first 10 ticks.

What the scheduler can ensure in the above case is that, at time equal to the least common multiplier (LCM) of all the processes periods, each process i will execute LCM/T_i times. For the above example, LCM = 20 ticks; so P1 will execute 10/20 = 1 times and P2 will execute 20/10 = 2 times over an interval of 20 ticks. If the above condition is acceptable, this scheduling scheme is better than the old one because the most likely result with the old algorithm under a high processor utilization is that P2 would miss a deadline.

The aperiodic processes can also be integrated with the above scheme of scheduling. One difference between them and the periodic ones is the trigger that will cause the start of the execution. For the periodic ones, the trigger is the timing process. For the aperiodic ones, it is an external event that will happen and cause some data to be written in the pipe, which the aperiodic process is waiting to read. For example, in the AUV-II, the task that is processing the sonar data, if it decides that there is an obstacle, will write in the pipe *obstacle alert* and the process ***avoid obstacle*** will start execution.

Another difference in incorporating aperiodic processes with periodic ones is the way the priorities are assigned. An array is created for the periodic processes. The first process in that array is the one with the smallest period, and therefore, the highest priority. The last one is the one with the largest period, and therefore, the lowest priority [Ref. Le91]. For the aperiodic processes, since the average period in which those processes arrive is not of great importance, another criterion had to be found according to which they will be placed in the array and the priorities will be assigned.

```
for ( i=1 ; i=#of aperiodic processes ; i=i+1)
{
          find lowest priority for combined process(i);
          place the i th process in that position;
          remove the i process from the list;
}
```

Figure 3.6: Algorithm to assign priorities to aperiodic processes.

One approach is to give priorities to the aperiodic processes that are either higher or lower than the periodic ones. After running some sets of tasks with such assignment, it was found that some aperiodic processes with low timing requirements were blocking the execution of periodic processes without any reason. Also, aperiodic processes with high timing requirements were blocked from periodic ones with lower timing requirements. The solution is an algorithm to enable an aperiodic process to block only the required number of periodic processes in order to achieve the expected response time. The algorithm that has been used is shown in Fig. 3.6. In this algorithm, the term *combined process* refers to a hypothetical process with response time equal to the minimum response time of all the aperiodic processes that have equal or higher *laxity*. Its execution time is equal to the sum of the execution times of aperiodic processes with equal or higher laxity. Laxity is defined as the response time minus the execution time of a process. In the following example, an application of the algorithm in Fig. 3.6 is demonstrated.

Example: In this example, all the times are in ticks. Consider four periodic processes P1, P2, P3, and P4 with execution times C_1=1, C_2=3, C_3=1, and C_4=4. The periods are T_1=5, T_2=10, T_3=10, and T_4=20 respectively. Consider two aperiodic processes, A1 and A2, with execution times, C_{A1}=10 and C_{A2}=6, average interarrival

No	Name	Period [tick]	Exec.Time [tick]	Resp.Time [tick]
1	P1	$T_1 = 5$	$C_1 = 1$	
2	A1	$T_{A1} = 100$	$C_{A1} = 10$	$R_{A1} = 20$
3	P2	$T_2 = 10$	$C_1 = 3$	
4	A2	$T_{A2} = 100$	$C_{A2} = 6$	$R_{A2} = 30$
5	P3	$T_3 = 10$	$C_3 = 1$	
6	P4	$T_4 = 20$	$C_4 = 4$	

Figure 3.7: Order in which priorities will be assigned.

periods, T_{A1}=100 and T_{A2}=100, and maximum response times, R_{A1}=20 and R_{A2}=30. According to the rate monotonic algorithm, the periodic process P1 has the highest priority and P4 has the lowest. The priorities to be assigned to the aperiodic processes are determined as follows:

- Find the position where the combined aperiodic process for A1, which has response time R_{COM1}=20 and execution time C_{COM1}=16, meets the timing requirements. That position is between P1 and P2, because, in the worst case that P1, P2, and A1 start together at time interval R_{COM1}=20, P1 needs 4, P2 needs 6, and A1$_{COM}$ needs 16 ticks.
- Place A1 between P1 and P2.
- Find the position where the combined aperiodic process for A2, which is only A2, meets its timing requirements. This process can be placed between P2 and P3 for the same reason.
- Place A1 between P2 and P3.

Figure 3.7 displays the final list according to which the priorities are assigned. If the combined aperiodic process had not been used, A1 would have been placed between P2 and P3. Also, A2 would have been placed between P2 and A1. The order, in that case, would have been the one in Fig. 3.8. This order is not correct because A2 has a priority higher than A1. Therefore, A1 is not able to meet its response time if A2 is being executed.

No	Name	Period [tick]	Exec.Time [tick]	Resp.Time [tick]
1	P1	$T_1 = 5$	$C_1 = 1$	
2	P2	$T_2 = 10$	$C_1 = 3$	
3	A2	$T_{A2} = 100$	$C_{A2} = 6$	$R_{A2} = 30$
4	A1	$T_{A1} = 100$	$C_{A1} = 10$	$R_{A1} = 20$
5	P3	$T_3 = 10$	$C_3 = 1$	
6	P4	$T_4 = 20$	$C_4 = 4$	

Figure 3.8: Priority assignment not using combined process concept.

The called processes have the general form of Fig. 3.9. The first modification made in the called processes was to include the `open()` statment in the initialization commands so the process can have access to the pipes. Also, a call to `signal_to_run-time_scheduler()` is included to signal the scheduler that initialization is completed. The next modification was to include the `read()` part in the repeated commands. The read commands are different if the process tries to read the output of a periodic process as against that of an aperiodic process. If the process reads from a periodic one, it directly tries to read the pipe. So, it is suspended until the information is ready. If, on the other hand, it reads from an aperiodic process, the `read()` is included in an `if()` statement that first checks the pipe to see if there is any information ready, as in:

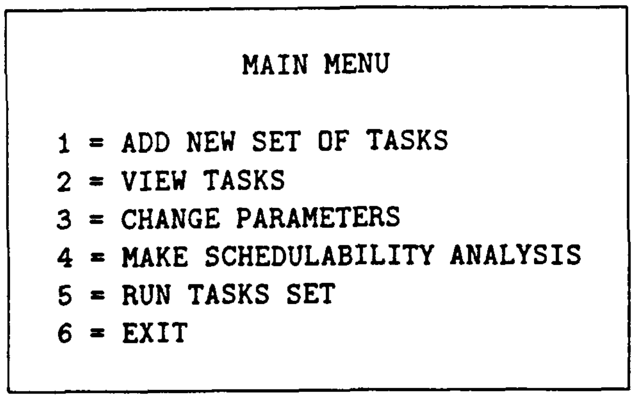
```
if( information at pipe ) then( read(pipe) );
```

This is done because it is not desirable to block the execution of a process for data that is not always expected to be there. Finally, the `write()` statement is included where the process sends the output to other processes.

Since pipes are files, the data transferred has to be characters. However, it can be formatted in any desired order. At the receiver, it can be modified, without restrictions, to any kind of variables like integers, floating point numbers, or strings.

```
main()
{
    initialization commands;
              .
              .
              .
    open(pipes);
    wake_up_run-time_scheduler();
    pause();
    while(TRUE){
        read(pipes);
        repeated commands;
              .
              .
              .
        write(pipes);
    }
    close(pipes);
    wake_up_run-time_scheduler();
    exit();
 }
```

Figure 3.9: General structure of a called process.



```
            MAIN MENU

1 = ADD NEW SET OF TASKS
2 = VIEW TASKS
3 = CHANGE PARAMETERS
4 = MAKE SCHEDULABILITY ANALYSIS
5 = RUN TASKS SET
6 = EXIT
```

Figure 3.10: Main menu for the run-time software.

For the purpose of collecting the timing data, a function that takes the time stamp has been included at the beginning and the end of the `while()` statment. Also, a delay loop has been included instead of the repeated commands with variable time length. That delay is now an input variable from the run-time scheduler. The source code of one of the dummy processes that was used is outlined in Appendix C.

C. USER INTERFACE

The run-time software has been completely changed in order to separate the initialization, the schedulability analysis, and the task execution part. The code of the run-time software is outlined in Appendix A and the code of the functions that are used by the run-time software is outlined in Appendix B. When the program starts, a menu appears on the screen similar to the one in Fig. 3.10.

By selecting `ADD NEW SET OF TASKS`, the program first asks the user if he/she desires to insert the information by using the keyboard or by having the program read a file. In the latter case, the user has to specify the name of the file. If the user decides to use the keyboard, the program first asks how many periodic tasks will be executed. Then, for each one of these, the user has to supply the name, the

```
           ----------- PERIODIC PROCESSES ----------
             NAME        PERIOD       EXECUTION TIME
           |  P1    |      5     |          1        |
           |  P2    |     10     |          3        |
           |  P3    |     10     |          1        |
           |  P4    |     20     |          4        |
           ------------------------------------------

  ----------------- APERIODIC PROCESSES ------------------
   NAME        PERIOD      EXECUTION TIME   RESPONCE TIME
  |  A1    |     100    |        10       |       20       |
  |  A2    |     100    |         6       |       30       |
  ---------------------------------------------------------
```

Figure 3.11: Display on the screen of the attributes.

period and the execution time. After the user has finished with the periodic processes, the program asks for the number of aperiodic processes. For each aperiodic process, the user has to specify the name, the average period, the execution time, and the maximum response time. All the above timing attributes have to be in ticks. When all the attributes have been inserted, the program sorts the periodic and aperiodic processes. The periodic processes are sorted in the increasing order of periods. The sorting for aperiodic processes is in the increasing order of laxity. The data-flow diagram that the process set implements cannot be changed by using the scheduler at this time. If the user wants to change it, he/she has to change the `pipes.c` program and the called processes.

By selecting `VIEW TASKS`, an output similar to the one in Fig. 3.11 appears on the screen. By selecting `CHANGE PARAMETERS`, the program first asks the user to specify if he/she wants to change a periodic or an aperiodic process. Then the user, following the on-screen instructions, can change the name and attributes of a process.

By selecting MAKE SCHEDULABILITY ANALYSIS, the run-time software executes two major objectives. Firstly, it makes the schedulability analysis for both periodic and aperiodic processes, and at the same time, fills an array with the names of the processes in the order discussed in Section B. Secondly, it creates an array with entries of the priorities that each process receives if the user later decides to run the set of tasks.

The schedulability analysis for the periodic processes is almost the same as the one explained by B Leatherman in the [Ref. Le91] and the software mainly consists of functions that are part of the rate_mono program in the same reference. If the periodic set is schedulable, the schedulability analysis program displays the remaining processor utilization that may be used for aperiodic processes. If the periodic set is not schedulable, it displays the processor utilization and returns to the main menu.

Schedulability analysis of the aperiodic processes is done in two steps. First, the program checks if the processor utilization that remains from the periodic set of tasks is sufficient for the aperiodic processes. If it is, the following message is displayed:

```
Total set SCHEDULABLE
```

If it is not, the program displays the following message and exits the schedulability analysis function:

```
Total set NOT schedulable
```

In case schedulable set, the software proceeds to the second step. Starting with the aperiodic process that has lowest timing requirements, it checks if the laxity of that process is larger than the sum of all other aperiodic execution times. If it is not, the process may not meet its response time in case that all aperiodic processes start together. In that case, the first message is augmented as:

```
But response time of "A2" process may NOT be achieved
```

By selecting RUN TASK SET, the run-time software starts execution of the set of tasks. This part is extremely dependent on the synchronization primitives that have been selected. First, it calls a function that forks the process `pipes.c` which creates all the pipes. This process remains active until all the tasks finish their execution; otherwise, if that process were allowed to finish execution, all the pipes would close. The same function also forks the process `start.c` that supplies the information that the set of tasks needs to start execution. The process `start.c`, at present, just sends the word "start" in a designated pipe. It can be easily modified to ask the user for the required initialization information. The code for `pipes.c` is outlined in Appendix D.

Following this, the scheduler forks all the processes one by one. It stops after forking each process until the forked process finishes the initialization part. This is necessary because the initialization part of each process is now longer than its average execution time. The part of the scheduler that sends the wake-up signals has been modified as has been explained in Section B.

After the run-time scheduler has forked all the processes, it waits in a loop with `pause()` till all the forked processes have completed execution. This ensures that the scheduler does not exit before all the forked processes have finished execution. Note that, in OS-9, a child process cannot exist without the parent process.

Finally, by selecting EXIT, the program provides to the user the option to save the task set names with all the attributes into a file for later use.

D. PERFORMANCE OF PIPE-BASED SYNCHRONIZATION

As described in the AUV-II data-flow diagram in Chapter I, it is likely to have eight processes. Six are periodic and two are aperiodic. The periodic processes are designed to run at 10Hz. By using the rate monotonic algorithm, when all the processes are independent, the expected processor utilization is equal to 1.0 according

Name	Period [tick]	Exec.Time [tick]
P1	$T_1 = 5$	$C_1 = 1$
P2	$T_2 = 10$	$C_1 = 2$
P3	$T_3 = 10$	$C_3 = 2$
P4	$T_4 = 40$	$C_4 = 8$

Figure 3.12: Process set with different execution frequencies for Experiment 1.

to [Ref. LL73]. When processes are running at different frequencies, the results are according to [Ref. Le91], with the difference that each independent set of tasks is treated as one process. We are interested in what the maximum processor utilization will be when pipe-based synchronization is included for the following situations:

- Only periodic processes are present.
- Both, periodic and aperiodic processes, are present.

Also for the second situation, it is of interest to verify that the response time of the aperiodic processes is acceptable.

E. EXPERIMENTAL RESULTS

The first set up used to verify the scheduler is one with four periodic processes. In Fig. 3.12, the attributes of those processes are displayed. Figure 3.13 displays the data-flow diagram for these processes. The number on the arcs describes how many items are produced and how many are consumed each time a process is executed. The above set yields a processor utilization very close to 1.0. The timing diagram of the process execution is displayed in Fig. 3.14. After the first complete execution, the set is periodic with period 40 ticks, which is the LCM of all the periods. Increasing the load delays the first execution of P4.

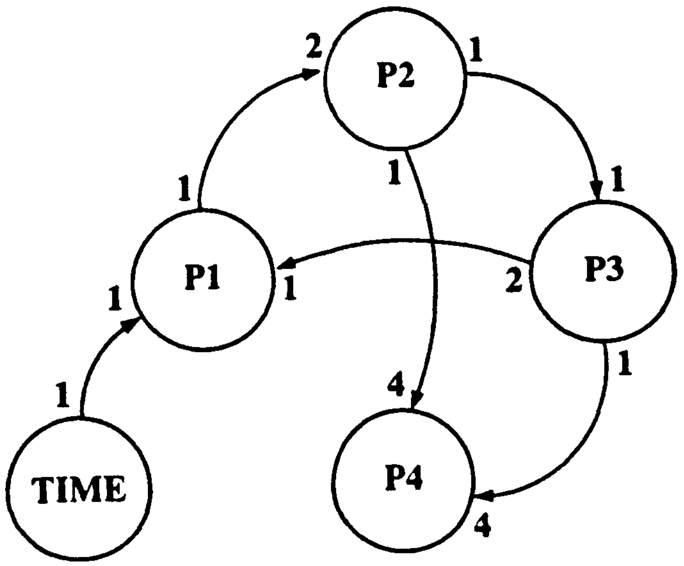


Figure 3.13: Flow diagram in experiment 1.

The second set used to verify the scheduler was one with periodic and aperiodic processes. This set uses the data-flow diagram in Fig. 3.15 and mimics the AUV requirements. The process set and the attributes that were used are shown in Fig 3.16.

The timing information obtained from this execution verified the following:

- The aperiodic processes meet all their timing requirements, if their interarrival time is the expected one.
- The processor utilization can be as high as 1.0.
- The periodicity of the periodic processes, although disrupted by the arrival of an aperiodic process, is restored eventually after a few cycles of execution.
- All the worst case conditions can be predicted in advance.

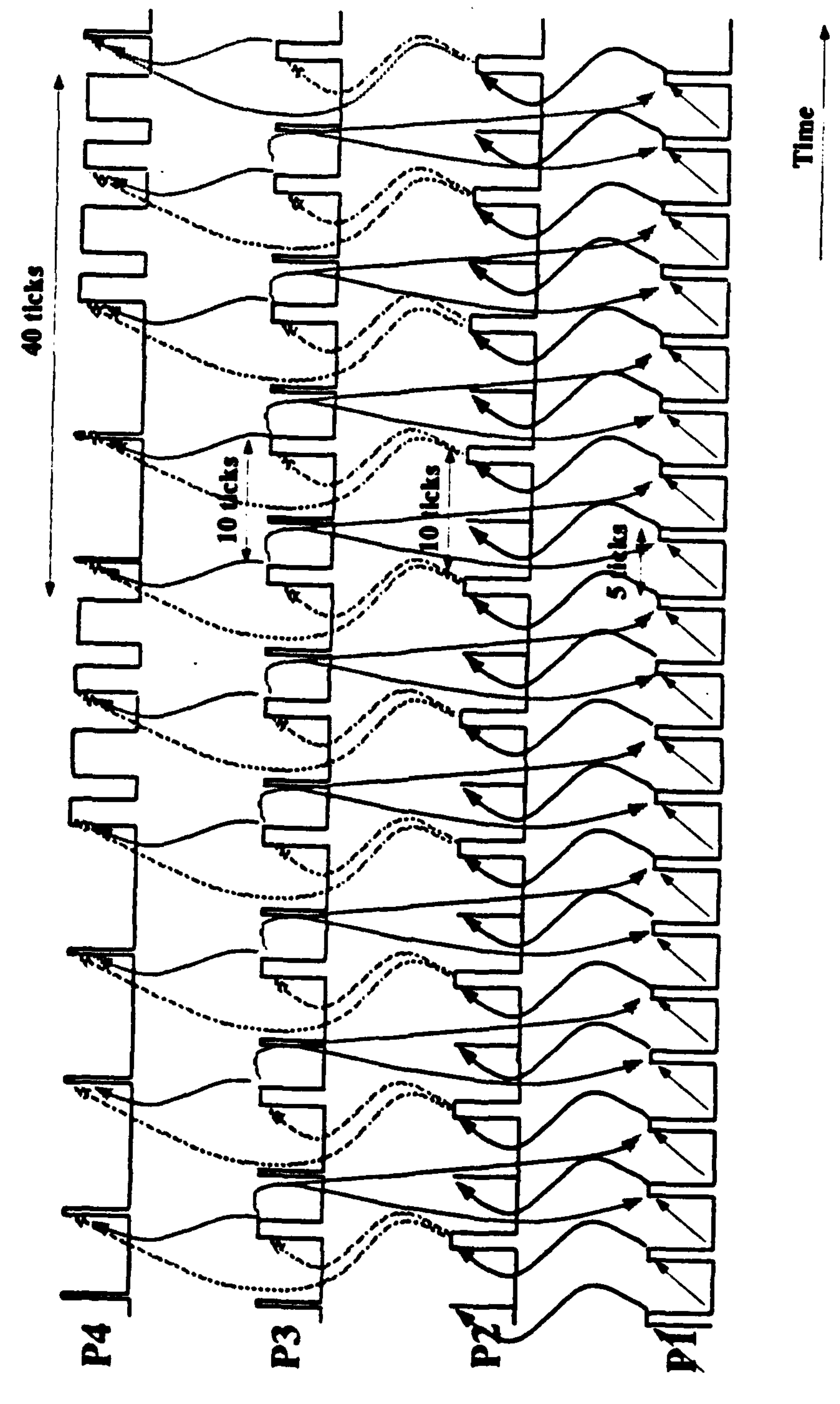


Figure 3.14: Timing diagram for experiment 1.

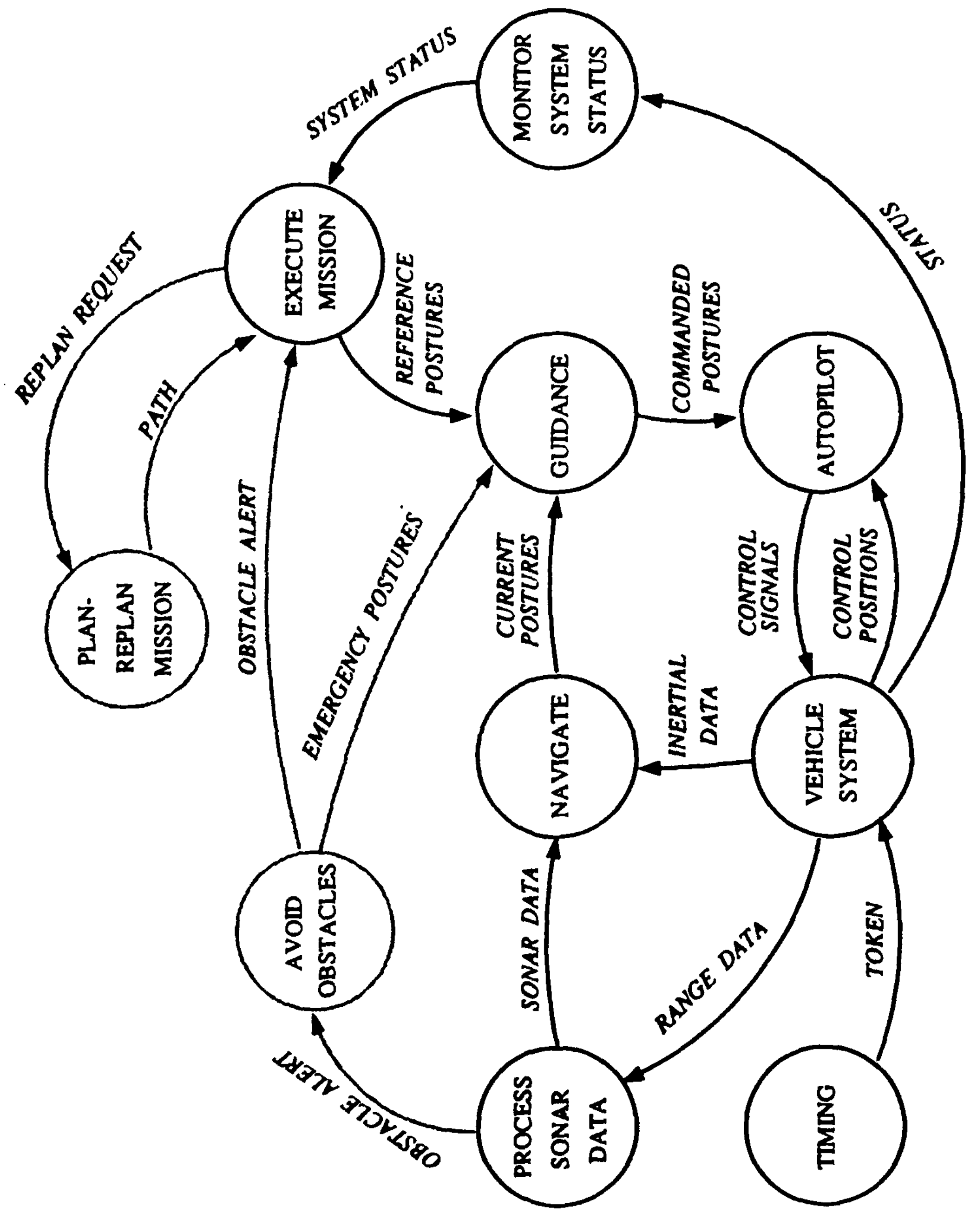


Figure 3.15: Processes set for experiment 2.

Name	Period [tick]	Exec.Time [tick]	Resp.Time [tick]
Vehicle system	$T_1 = 10$	$C_1 = 1$	
Sonar	$T_2 = 10$	$C_2 = 1$	
System status	$T_3 = 10$	$C_3 = 1$	
Navigate	$T_5 = 10$	$C_5 = 1$	
Avoid obstacle	$T_4 = 500$	$C_4 = 8$	$R_4 = 20$
Execute mission	$T_6 = 10$	$C_6 = 1$	
Guidance	$T_7 = 10$	$C_7 = 1$	
Autopilot	$T_8 = 10$	$C_8 = 1$	
Plan mission	$T_9 = 500$	$C_9 = 4$	$R_9 = 100$

Figure 3.16: Order in which priorities are assigned.

IV. FRAMEWORK FOR A GRAPHICAL USER INTERFACE

A. GRAPHICAL USER INTERFACE REQUIREMENTS

1. Current Operation

The AUV-II scheduler, as described in Chapter III, performs schedulability analysis for periodic and aperiodic processes before run-time and performs scheduling and synchronization at run-time. However, the availability of a schedulability analyzer and a run-time scheduler does not make the cycle of real-time software development easy. The user of these tools, who is unlikely to be an expert in real-time scheduling, must find them easy to use. Therefore, there is a need for a Graphical User Interface (GUI) capable of accomplishing these functions in an easy way. The current user interface communicates with the user by asking questions and receiving appropriate answers.

The information that this simple interface can supply to the user, through the main menu, is the display of the last selected set of tasks and their attributes. After the analysis for a given set of tasks has been completed, the software supplies to the user the processor utilization for periodic tasks, prediction of whether or not the periodic set is schedulable, total processor utilization, prediction of whether or not the total set is schedulable, prediction of aperiodic processes that may not meet their timing requirements, and finally, order in which the priorities will be assigned. Also, the user can manually find the time intervals at which each process is executed by reading the output files that are created for that purpose from each task.

The above information appears on the screen once, stays there as long as the user wants to inspect it, and then disappears. This causes difficulties when the user has to correlate more than one pieces of information at the same time. The only available solution to this problem is manual tracking. For a large number of tasks, such handling of information is difficult and error prone. This condition cannot be improved beyond a limit because the user interface was designed in the old fashion question-answer process. This interface is also limited by the capabilities of the currently used display hardware which is a simple VT220 terminal.

2. Requirements

Since the AUV-II is still in its development stage, one or more of the following scenarios are likely to occur:

- A process can be omitted or combined with other processes. New processes may be required for the vehicle operation.
- For any task, more than one versions may be written to implement different algorithms. This will force the user to select from a collection of tasks each time.
- More than one scheduling algorithm may be implemented. Currently, the user has a choice of selecting the scheduling algorithm created in [Ref. Le91], or the one proposed in Chapter III, or the OS-9 scheduler without any enhancement.
- Process attributes may be change because of modifications in the programs.
- After one pass of the schedulability analysis, the user may need to change task attributes in order to find a workable combination for the application.

Also, the scheduling scheme that has been proposed can be used, not only by the AUV-II project, but by other real-time applications also.

Given these likely scenarios and the fact that a replacement of the current computer by a portable workstation is planned, it was decided that a Graphical User Interface (GUI) would make the AUV-II software development process much simpler. Using a mouse and a keyboard, this GUI will enable the user to perform the following tasks easily:

- Select a real-time application.

- Select a scheduling algorithm.
- Select the tasks that he/she wants to use.
- Modify the attributes of a task.
- Have all the selected tasks displayed.
- Have the results of the schedulability analysis displayed.

GUI's are, in general, difficult to implement because the designer has to take care of more than one input device like the mouse and the keyboard at the same time. This implementation is somewhat simplified by the availability of standard software packages for manipulating the display. However, the designer must pick from hundreds of procedures in every package in order to achieve his goal. One system that provides the necessary tools to a GUI designer is the X window system that is overviewed in the next section.

B. X-WINDOW SYSTEM

The X Window System is a software environment which is used for engineering workstations [Ref. Jo89]. It has the capabilities to control the displays and to provide a standard environment for different applications.

The X environment consists of layers built upon the *base window system*, as can be seen in Fig. 4.1 [Ref. Jo89]. The base window system is able to have outside communication by using the X network protocol, which is also the only way to communicate with it.

Because it is hard to use the network protocol directly, there is a low-level programming interface named Xlib. This is a package containing subroutines in C language. There are also higher level toolkits by which the details of the network

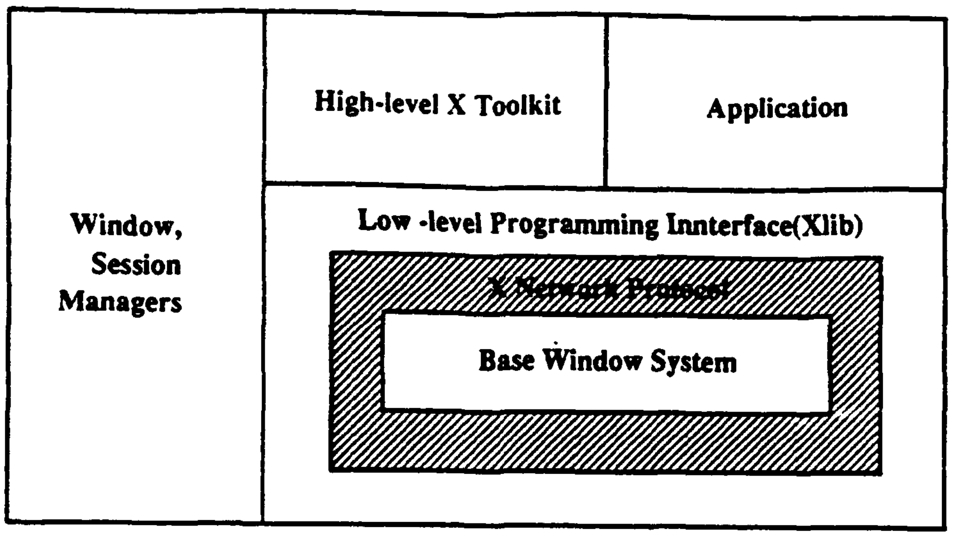


Figure 4.1: X software environment.

protocol are masked. Xlib provides a way to receive for different inputs in the application programs, such as inputs by pressing keys on the keyboard or by moving and pressing the mouse buttons. It also provides tools for output that has the following capabilities.

- The screen can be organized in a hierarchical fashion by overlapping windows, resizing them, moving them, and putting as many as the application needs on top of each other.
- Each drawing is bitmapped and corresponds to a specific address on the specific window.
- High-quality text can be sent to the screen.
- A wide variety of colors, as well as black and white models, can be used.
- Manipulation of different images is possible.

Since Xlib allows the designer to access the above facilities, it provides a rich environment for building user interfaces. As low level environment, it allows direct control of all the details required by the application. For these reasons, it was selected as the most appropriate tool to create a GUI for the AUV-II.

As has been mentioned earlier, the way that the application communicates with the base window system, and vice versa, is via a network protocol. The fundamental elements of this protocol are *requests* and *events*. Requests are messages that originate from the application and events are messages that originate from the workstation.

The request messages instruct the workstation to take actions required by the application such as, opening a window, changing the color, displaying some context, etc. On the other hand, event messages are used for external events that affect the application, such as a movement of the cursor, a keyboard button press, a press of the mouse, etc. How the application interprets all these events is determined only by the designer who writes the appropriate code in the application program.

A GUI creates, uses, and destroys different *resources* in the course of its operation. These resources are the tools that make the user interface friendly. The resources that were used in this design are:

- *Windows*: These are rectangular areas on the video screen. In every window, the information that the user would like to be visible is specified. These windows may overlap one another according to the user's input.
- *Graphical Contexts*: These resources are used to specify the style, size, line width for the text that the user wants to be displayed, foreground and background colors, etc.
- *Fonts*: These resources are the ones that control the character text that are used, like shape, size, etc.
- *Pixmaps*: These resources are used by the application to copy information between windows. This capability is very helpful since all the selections, corrections, additions, etc., may not take place on the main window, but on others that may open according to the user's will.
- *Cursors*: With these resources, the cursor shape of the on-screen pointer can be manipulated which the user can move around by moving the mouse.

C. GUI IMPLEMENTATION

By using Xlib, a GUI consisting of a main window in which all the required information is displayed and some pop-up windows that could be used for different selections and modifications was created.

1. Main Window

In the main window, as can be seen in Fig. 4.2, the present status of the process set, together with the selection buttons and the output of the schedulability analyzer, is displayed.

At the top of the window, there is the title **SCHEDULER FOR:** followed by the object to be analyzed. This reflects that the GUI can be used in applications other than the AUV-II. The present status of the tasks is displayed in the two windows described below.

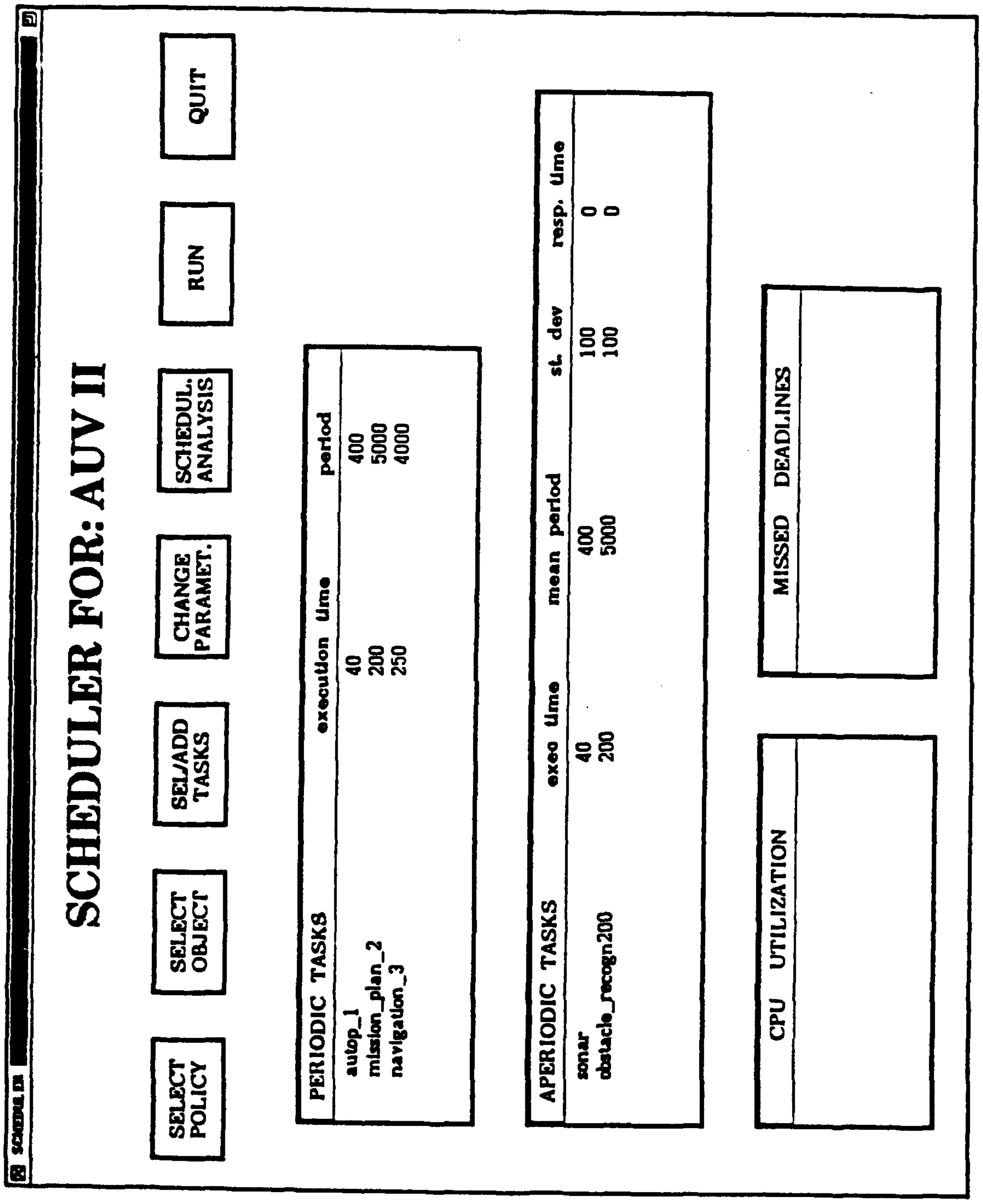
- PERIODIC TASKS: In this window, the user can observe all the selected periodic tasks, their execution time, and their period. This information is also needed by the schedulability analyzer.
- APERIODIC TASKS: In this window, the user can observe the selection for the aperiodic tasks, their execution time, mean period, standard deviation from the mean period, and the maximum response time in which the process is expected to finish execution.

Output of the schedulability analyzer is displayed in two windows as described below.

- CPU UTILIZATION: It contains information for the CPU utilization for periodic tasks, aperiodic tasks, and the total set of tasks.
- MISSED DEADLINES: In this window, the user can observe if the deadline of a process was missed and which process missed it.

On top of the main window are the selection buttons. Any of them is selected by placing the cursor on top of one and clicking the mouse. The use of textual buttons was preferred instead of icons because it is very difficult to have a design representing the required meaning. The selection buttons are the following:

- SELECT POLICY.
- SELECT OBJECT.
- SELECT/ADD TASKS.



PERIODIC TASKS	execution time	period
autop_1	40	400
mission_plan_2	200	5000
navigation_3	250	4000

APERIODIC TASKS	exec time	mean period	st. dev	resp. time
sonar	40	400	100	0
obstacle_recogn200	200	5000	100	0

Figure 4.2: Main window.

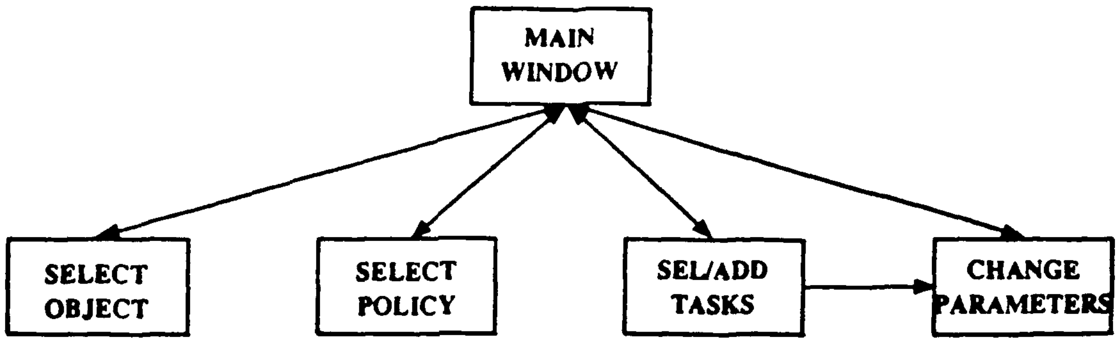


Figure 4.3: GUI window diagram.

- CHANGE PARAMETERS.
- SCHEDULABILITY ANALYSIS.
- RUN.
- QUIT.

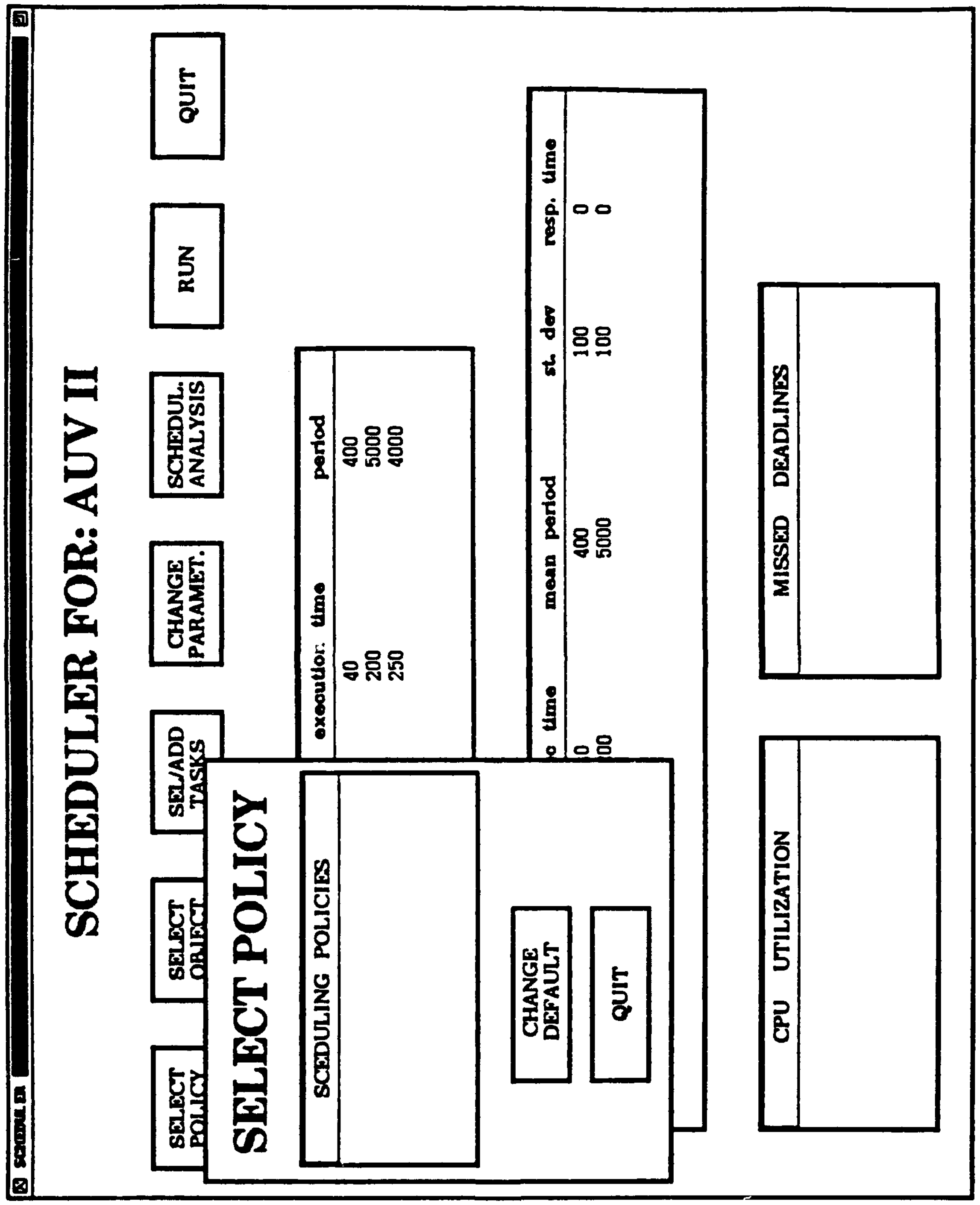
The results of selecting each one of these buttons are listed in the next section.

2. Pop-up Windows

The first four buttons open the pop-up windows and the following three call the appropriate function. Figure 4.3 displays how the user can move around the windows.

By selecting the button **SELECT POLICY**, the window in Fig. 4.4 opens on top of the main one. With this, the user can select the scheduling policy that he/she wants to use in order to execute the set of tasks or to perform schedulability analysis.

By selecting the button **SELECT OBJECT**, a window similar to the one in Fig. 4.4 opens. With this option, the user can select the object for which this user interface will be used. Also, the title on top of the main window changes. In both of the above windows, the different options are read from a file.



execution time	period
40	400
200	5000
250	4000

o time	mean period	st. dev	resp. time
0	400	100	0
00	5000	100	0

Figure 4.4: Select policy window.

By selecting the button **SEL/ADD TASKS**, the window in Fig. 4.5 appears. This window consists of two subwindows – one for periodic tasks and the other for aperiodic ones. Since it is possible to have more than one programs for each task, the user can select the ones that he/she wants to use for each experiment. In this window, the scroll bars can be used to change the selections, the keyboard to add new ones, and the mouse to either change the default ones. The mouse can also be used to go to the window CHANGE PARAMETERS and simultaneously close the present one or to quit this window and go to the main one.

By selecting the button **CHANGE PARAMETERS**, the window in Fig. 4.6 appears. It is the one that the can be used to give the tasks attributes needed by the schedulability analyzer. This window also consists of two subwindows –one for the periodic and the other for the aperiodic tasks.

By selecting the button **SCHEDULABILITY ANALYZER**, according to the selected policy in the SELECT POLICY window, the software calls the function that implements the specified schedulability analysis and supplies the information in the windows CPU UTILIZATION and MISSED DEADLINES.

By selecting the button **RUN**, the software calls the function that starts the execution of the selected tasks.

Finally, the button **QUIT** quits the scheduler and closes the main window.

D. PROBLEMS IN THE GUI IMPLEMENTATION

The following problems arose in developing the GUI. First, at the time of the GUI execution, some windows did not appear on the screen or sometimes they could not receive an event such as a button press. This was due to an excessive number of open windows that were created. As an example, in Fig. 4.5, the window that displays the periodic tasks consists of the window that is displayed, a larger window below the

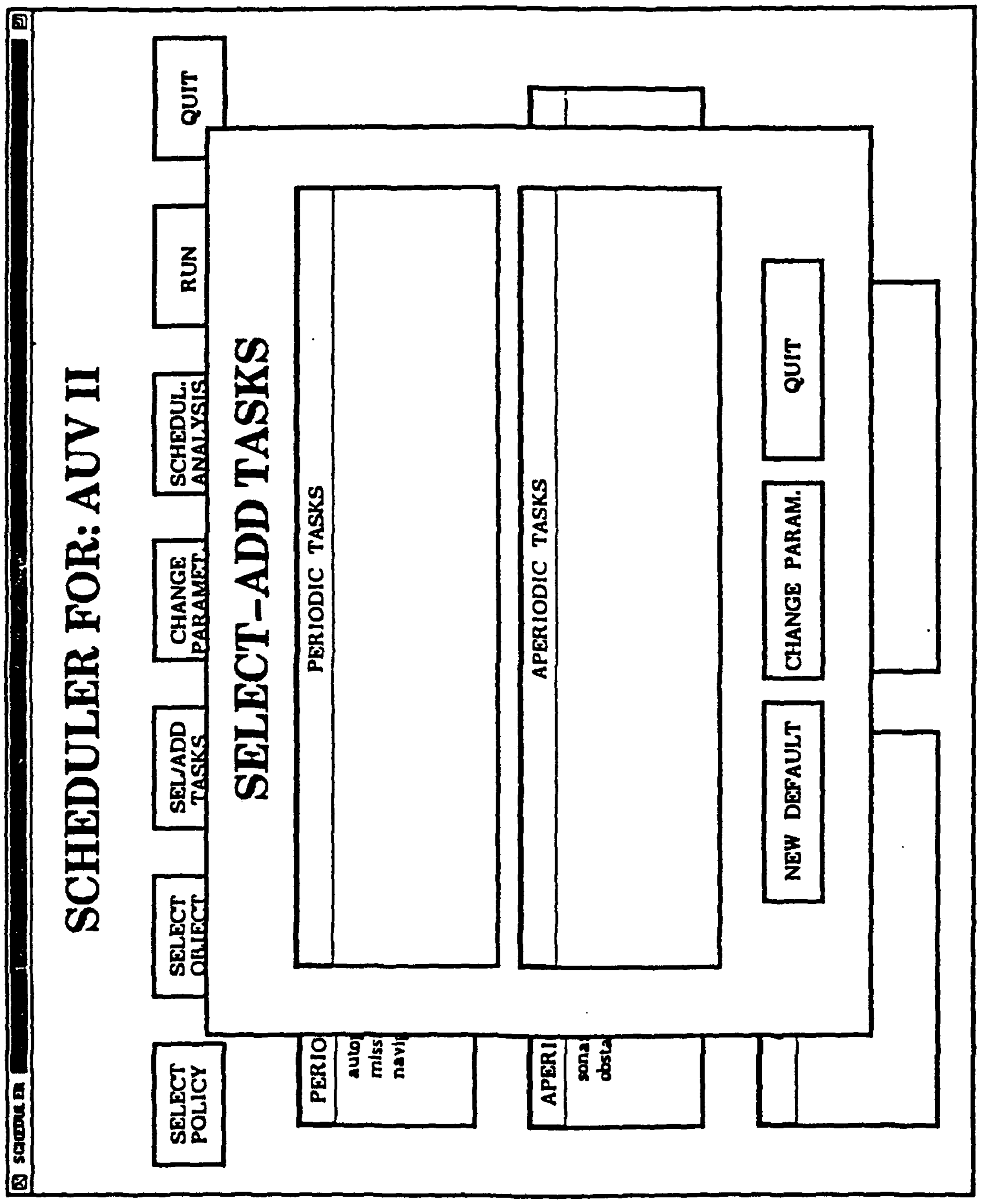


Figure 4.5: Select/Add tasks window.

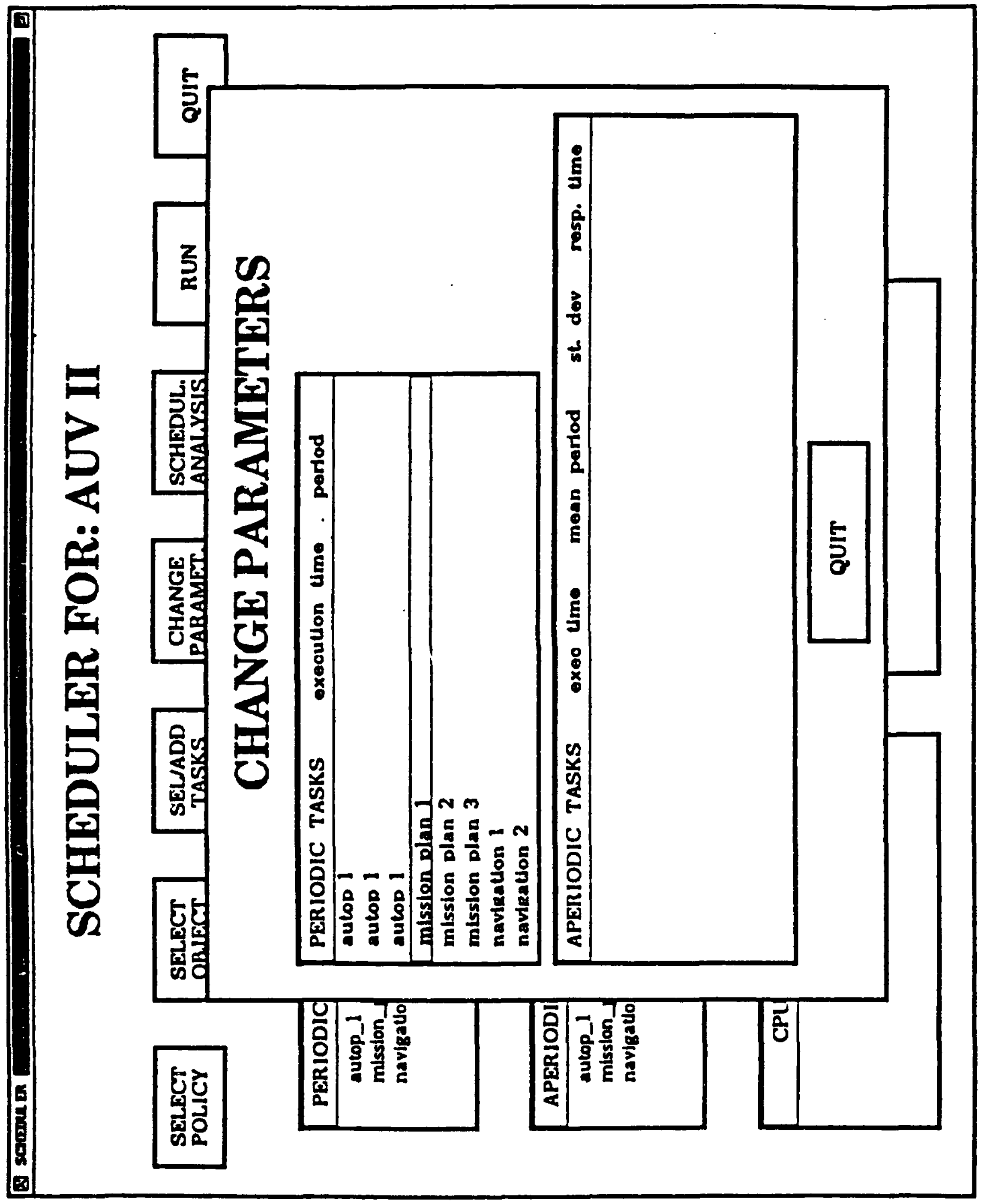


Figure 4.6: Change parameters window.

one that appears, and an array of windows in which each window element displays the attributes of one task. The reason for having a large window is to provide the user with the ability to scroll the text up and down. The reason of having an array of windows, one for each task, is to provide the capability to select only one task at a time. In order to solve this problem, more interactive communication between the application, the window manager, and the memory manager is needed.

The next problem was the use of the keyboard to add or select some of the tasks' attributes. Unlike a simple video display terminal where whatever the user types is displayed on the terminal, in an X window application, when the user presses a key, an event is sent to a specific window. It is then up to the application to interpret that event. For modifying task attributes, an entire editor has to be created.

One solution to most of the above problems is the X toolkits whose purpose is to simplify the GUI programming. One X toolkit that provides all the required components for a GUI is the Athena widget set. The Athena X widgets consist of a set of prebuilt windows with special characteristics that can be used as components to create a GUI. Some of those widgets are menus, dialogue boxes, scrollbars, text widgets, etc. Our experience in building a GUI using Xlib is aptly summarized by quoting Mark Langley from [Ref. NO90, page 37]:

> Window systems may be simple to use, but they are very complex to program. The first thing that strikes the novice X programmer is how complicated everything is. Learning to program the X Window System, even with the help of the X Toolkit, is a far cry from learning say, the C programming language...

V. CONCLUSIONS AND FUTURE DIRECTIONS

This chapter summarizes what the study has achieved for the development of the AUV-II real-time software and provides some directions for further research and improvements.

A. CONCLUSIONS

The major objectives of this study, as have been specified in Chapter I, have been accomplished. A scheme capable of scheduling both periodic and aperiodic processes has been created. Periodic processes are assigned priorities according to the rate monotonic scheduling. Aperiodic processes are assigned priority in order to achieve the expected response time, and at the same time, block as few periodic processes as possible. Thus, the scheduling scheme yields a high processor utilization and handles both kinds of processes smoothly by providing the required frequency in the periodic ones and the expected response time in the aperiodic ones.

The synchronization primitive that has been selected is OS-9 PIPE. Its use provides, without any additional programming, both mutual exclusion and data consistency. Although synchronization conflicts with scheduling, this scheme achieves a utilization close to 1.0 for the AUV-II application which has a single independent set of processes.

The usefulness of the scheduler for the AUV-II has been verified by creating a set of processes that mimics the operation of the AUV-II. This set is executed at the required frequency of 10 Hz and, at the same time, the expected response time for the aperiodic processes is achieved. This dummy set of processes can be applied in the AUV-II if the idle loop in each process is replaced by a call to an appropriate function and the inputs and outputs on pipes are replaced with the real data to be transferred. Finally, a framework for a GUI was designed and experimented with to provide the functionality required for easy use of the scheduler.

B. FUTURE WORK

At present, the synchronization to be provided has to be coded manually as described in Chapter III. This coding is required for pipe initialization in the process that is being called from the scheduler and in the application processes. This can be further improved in such a way that the user only has to specify the names of the different pipes, the processes that the pipes have to connect to, and explicitly specify which of those processes are periodic and aperiodic. An approach to implement this is to pass the pipes' names as arguments to the different processes, followed by a flag that determines if the input is from an aperiodic or a periodic process.

The schedulability analysis that has been provided refers only to the processes timing requirements and does not make any analysis of the data-flow synchronization. Thus, the scheduler cannot identify inconsistencies in the data-flow diagram. Such analysis has to be included to ensure a correct application of the synchronization technique developed. Techniques that can be used for data-flow analysis are provided in [Ref. LA90].

Finally, if the above facilities are included in the scheduler, the framework of the GUI can be improved to become more user friendly. The new design, instead of displaying the names and the attributes of the different processes, can display the data-flow diagram that those processes implement. Each process can be a node that displays the name and the attributes. The pipes can be represented as arcs between the nodes. Those arcs can produce the required arguments that will be given to the processes. The periodic and aperiodic processes can be distinguished at this time appropriately.

REFERENCES

[Ba90] T.P. Baker, "*A Stack-based Resource Allocation Policy for Real-time Processes*", Proc. of IEEE Real-Time Symposium, pp. 191-200, 1990.

[Cl90] Michael John Coutier, "*Guidance and Control System for an Autonomous Vehicle*", Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

[Di88] P. Dibble, "*OS-9 Insights - An Advanced Programmers Guide to OS-9/68000*", Microware Systems Corporation, 1988.

[Fl91] Charles A. Floyd, "*Design & Implementation of a Collision Avoidance System for the NPS Autonomous Underwater Vehicle (AUV-II) Utilizing Ultrasound Sonars*", Master's Thesis, Naval Postrgraduate School, Monterey, CA, September 1991.

[GES88] "*GESMOS-68 OS-9/68000 Operating System*", Ver. 2.3, GESPAC, Inc., Geneva, SA, 1988.

[GES90] "*Microware OS-9/68000 C Compiler User's Manual*", Rev. H, Microware Systems Corporation, Des Moines, IW, October 1989.

[Jo89] Oliver Jones, "*Introduction to the X Window System*", PRENTICE-HALL, Ineglewood Cliffs, NJ, 1989.

[LA90] Shem Ton Levi and Ashok K. Agrawala, "*Real-time System Design*", McGraw Hill, USA, 1990.

[Le91] B. Leatherman, "*An Approach to Investigate of Real-time Software for the Autonomous Underwater Vehicle*", Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1991.

[LL73] C. L. Liu and J. W. Layland, "*Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment*", JACM 20, pp. 46-61,1973.

[LSD89] John Lehoczky, Lui Sha and Ye Ding, "*The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*", Proc. of IEEE Real-time System Symposium, pp. 166-171,1989.

[NO90] Nye Adrian and O'Reilly Tim, "*X Toolkit Intrinsics Programming Manual*", The X Window System Vol. 4, O'Reilly & Associates, Inc., Sebastopol, CA, September 1990.

[SG90] Lui Sha and John B. Goodenough, "*Real-time Scheduling Theory and ADA*", IEEE Computer, pp. 53-62, April 1990.

[SRL91] Lui Sha, Ragunthan Rajkumar and John P. Lehaczky, "*Priority Inheritance Protocols: An Approach to Real-time Synchronization*", IEEE Transaction on Computers, vol 39, No 9, September 1990.

[Ta87] Andrew S. Tanenbaum, "*Operating Systems Design and Implementation*", PRENTICE-HALL, INC. , Englewood Cliffs, NJ, 1987.

[TK88] Hideyaki Tokuda and Makoto Kotore, "*A Real-time Tool Set for ARTS Kernel*". Proc. of IEEE Real-ime System Symposium, pp. 289-299, 1988.

APPENDIX A: MAIN FOR RUN-TIME SCHEDULER

```
/*********************************************************************
*   Program     :  SCHED.C                                           *
*   Purpose     :  MAIN CODE FOR AUV-II SCHEDULER.                   *
*   Author      :  LTJG D. MAKRIS  H.N.                              *
*   Description:  THIS PROGRAM, IN AN INFINITE LOOP, CALLS THE MENU  *
*                 FUNCTION. ACCORDING TO THE USER SELECTION, AN      *
*                 APPROPRIATE FUNCTION IS CALLED.                    *
*                 THE SELECTIONS AVAILABLE TO THE USER ARE:          *
*                 1. ADD NEW SET OF TASKS                            *
*                 2. VIEW TASKS                                      *
*                 3. CHANGE TASKS PARAMETERS                         *
*                 5. START EXECUTION OF THE TASK SET                 *
*                 6. EXIT THE SCHEDULER                              *
*********************************************************************/
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <procid.h>
#include <setsys.h>
#include <signal.h>
#include <ctype.h>

#define    TRUE         1
#define    NO           0
#define    DONE         1
#define    SIZE         15
#define    MAX_PRIORITY 60000

extern     int          kill(),os9forkc(),exit(),intercept();
extern     int          getpid(),_get_process_desc();
extern     double       atof();
extern     char         **environ;

                  /* THE FOLLOWING ARRAYS CONTAINS ALL THE ATRIBUTES IN
                     CHARACTER FORM SO THEY CAN BE PASSED AS ARGUMENTS
                     INTO THE PROCESSES THAT WILL BE FORKED.        */
char       *arg1[SIZE],*arg2[SIZE],*name[SIZE],
           *Cchar[SIZE],*Cchar_ap[SIZE],*tot_Cchar[SIZE],
           *Tchar[SIZE],*Tchar_ap[SIZE],*tot_Tchar[SIZE];
char       *arg3[]={"start",0,};
char       *arg4[]={"pipes",0,};
double     W[SIZE][SIZE],Li[SIZE][SIZE];

icpthand(signum)
```

```
int signum;
{
                            /* INTERCEPT HANDLER                    */
   fprintf(stderr,"I received signal in sched.c : %d\n",signum);
}

main ()
{
   double      T[SIZE],T_ap[SIZE],tot_T[SIZE],
               C[SIZE],C_ap[SIZE],tot_C[SIZE],
               D_ap[SIZE];
   int         n,aper_no,tot_no,analysis;
   int         process_pri[SIZE],pid;
   procid      parent;
   char        job;

   char        menu();
   void        create_memory();
   void        add_new_set();
   void        view_tasks();
   void        change_parameters();
   void        make_sched_analysis();
   void        run_set();
   void        stop_screen();
   void        my_end();

   analysis=0;
   pid=getpid();            /* FIND PROCESS ID                        */
                            /* FIND NEWSS (START SCHEDULER) ID        */
   _get_process_desc(pid, sizeof(parent),&parent);
   create_memory();         /* ALLOCATE MEMORY IN ALL ARRAYS          */
   while(TRUE)
   {
      job = menu(1);        /* CALL MANY #1 AND GET USERS ORDER       */
      switch(job){

         case '1':
            analysis=0;     /* SCHED. ANALYSIS HAS'T BE DONE          */
                  /* CALL FUNCTION THAT TAKE NEW PROCESSES VALUES */
            add_new_set(&n,&C[0],&T[0],
               &aper_no,&C_ap[0],&T_ap[0],&D_ap[0],
               &tot_C[0],&tot_T[0]);
            break;

         case '2':
                        /* CALL FUNCTION THAT DISPLAYS ATTRIBUTES */
            view_tasks(n,&C[0],&T[0],
               aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
            break;

         case '3':
            analysis=0;     /* SCHED. ANALYSIS HAS'T BE DONE        */
                      /* CALL FUNCTION THAT CHANGE TASKS ATTRIBUTES */
            change_parameters(n,&C[0],&T[0],
                aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
            break;

         case '4':
            analysis=DONE;
            tot_no =n;
                      /* CALL FUNCTION THAT MAKES SCHEDUL. ANLYSIS  */
            make_sched_analysis(&C[0],&T[0],&tot_no,
                 &C_ap[0],&T_ap[0],&D_ap[0],aper_no,&process_pri[0],
                 &tot_C[0],&tot_T[0]);
            break;

         case '5':
            if(analysis==DONE) /* IF SCHED. ANALYSIS HAS BE DONE    */
                         /* CALL FUNCTION THAT STARTS THE EXECUTION */
               run_set(&tot_C[0],&tot_T[0],
                     &process_pri[0],&parent,tot_no);
            else{            /* ELSE GO TO MAIN MANU                 */
             printf(" YOU MUST FIRST MAKE SCHEDULABILITY ANALYSIS\n");
               stop_screen();
            }
            break;

         case '6':
                    /* CALL FUNCTION THAT KILLS THE NEWSS AND EXITS */
            my_end(n,aper_no,&D_ap[0],parent);

         default:
            printf("\n\t This is not a proper command! Try again.\n");
            stop_screen();
      }
   }
}
```

APPENDIX B: FUNCTIONS USED BY RUN-TIME SCHEDULER

```
/**********************************************************************
*   Program     :  SCHED.C                                             *
*   Purpose     :  FUNCTIONS THAT ARE CALLED FROM main() IN THE        *
*                  AUV-II SCHEDULER.                                   *
*   Author      :  LTJG D. MAKRIS  H.N.                                *
*   Description:   THIS CODE CONTAINS ALL THE FUNCTIONS THAT ARE       *
*                  BEING CALLED IN THE AUV-II SCHEDULER.               *
*                  EACH FUNCTION WILL BE DESCRIBED SEPARATELY.         *
**********************************************************************/

/**********************************************************************
*   THIS FUNCTION PROVIDES ALL THE MENUS THAT ARE USED FROM THE        *
*   SCHEDULER. IT HAS AS INPUT THE NUMBER OF THE MENU THAT HAS TO      *
*   BE DISPLAYED AND RETURNS THE USER'S SELECTION FROM THAT MENU.      *
**********************************************************************/
char    menu(chose)
int     chose;
{
   int        job;
   char       bufer[5];

   printf("\n\n\n\n\n\n\n\n\n");
   switch (chose){
      case 1:                  /* DISPLAYS MAIN MENU                           */
         printf("\t\t\t\t\t          MAIN MENU      \n\n");
         printf("\t\t\t\t\t1 = ADD NEW SET OF TASKS  \n");
         printf("\t\t\t\t\t2 = VIEW TASKS            \n");
         printf("\t\t\t\t\t3 = CHANGE PARAMETERS     \n");
         printf("\t\t\t\t\t4 = MAKE SCHEDULABILITY ANALYSIS\n");
         printf("\t\t\t\t\t5 = RUN TASKS SET         \n");
         printf("\t\t\t\t\t6 = EXIT         \n");
         break;

      case 2:                  /* MENU WHEN THE USER SELECTS 1 IN THE
                                  MAIN MENU                                    */
         printf("\n\n\n\n");
         printf("\t\t\t\t\t1 = INPUT FROM KEYBOARD \n\n");
         printf("\t\t\t\t\t2 = INPUT FROM FILE \n\n\n");
         break;

      case 3:                  /* MENU WHEN THE USER SELECTS 2 IN THE
                                  MAIN MENU                                    */
         printf("\n\n\n");
         printf("\t\t\t\t\t1 = CHANGE PERIODIC SET \n\n");
```

```
        printf("\t\t\t\t\t2 = CHANGE APERIODIC SET \n\n\n");
        break;
  }
  printf("\n\n\n\n\n\n\n\n");
  readln(0,bufer,2);
  return(bufer[0]);
}

/**********************************************************************
*    THIS FUNCTION ALLOCATES MEMORY FOR ALL THE STRING ARRAYS       *
**********************************************************************/
void    create_memory()
{
   int        i;

   for(i=0;i<SIZE;i++){         /* FOR MAX. EXPECTED # OF PROCESSES */
      arg1[i]      = (char *)malloc(15);/* PERIODIC NAMES            */
      arg2[i]      = (char *)malloc(15);/* APERIODIC NAMES           */
      name[i]      = (char *)malloc(15);/* ALL NAMES                 */
      Tchar[i]     = (char *)malloc(5); /* PERIODIC PERIODS          */
      Cchar[i]     = (char *)malloc(5); /* PERIODIC EXEC. TIME       */
      Tchar_ap[i]  = (char *)malloc(5); /* APERIODIC AVER. PERIODS   */
      Cchar_ap[i]  = (char *)malloc(5); /* APERIODIC EXEC. TIMES     */
      tot_Tchar[i] = (char *)malloc(5); /* ALL PERIODS               */
      tot_Cchar[i] = (char *)malloc(5); /* ALL EXECUTION TIMES       */
   }
}

/**********************************************************************
*  THIS FUNCTION OBTAINS THE NEW SET OF TASKS THAT THE USER PROVIDES.*
*  FIRST ASKS THE USER TO SELECT IF HE WANTS TO ADD FROM A FILE OR   *
*  BY USING THE KEYBOARD. THEN CALLS THE APPROPRIATE FUNCTIONS TO    *
*  READ THE NAMES AND THE ATTRIBUTES. FINALLY MAKES SORTING FOR      *
*  BOTH PERIODIC AND APERIODIC PROCESSES.                            *
**********************************************************************/
void    add_new_set(n,C,T,aper_no,C_ap,T_ap,D_ap,tot_T,tot_C)
int     *n,*aper_no;
double  *C,*T,*C_ap,*T_ap,*D_ap,*tot_T,*tot_C;
{
   char       job;

   void       take_initial_values();
   void       short_periodics();
   void       short_aperiodics();
   void       take_aperiodic_values();
   void       take_from_file();

   job        = menu(2);    /* INPUT FROM FILE OR KEYBOARD ??        */
   switch(job){
      case '1':             /* FROM KEYBOARD                         */
         take_initial_values(n,&C[0],&T[0]);
         short_periodics(*n,&C[0],&T[0]);
         take_aperiodic_values(aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
         short_aperiodics(*aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
         break;
      case '2':             /* FROM FILE                               */
         take_from_file(n,&C[0],&T[0],
                        aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
         short_periodics(*n,&C[0],&T[0]);
         short_aperiodics(*aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
         break;
      default:
         printf("\n\t\tNOT a proper command! \n\n");
         stop_screen();
   }
}


/*********************************************************************
*     THE FOLLOWING FUNCTION RETURNS TO THE PROGRAM THE AVERAGE      *
*  TASK PERIODS T[], THE EXECUTION TIME C[] AND THE NAMES arg1[] OF  *
*  THE PERIODIC PROCESSES. FIRST ASK THE USER FOR THE NUMBER OF      *
*  PERIODIC PROCESSES THAT THE SET WILL HAVE AND THEN CALLS A        *
*  FUNCTION TO HAVE THOSE VALUES ONE AT THE TIME.                    *
*********************************************************************/
void    take_initial_values(n,C,T)
int     *n;
double  *C,*T;
{
   int        i;
   void       one_periodic();

   printf("\n\nNumber of periodic processes to schedule =>:");
   scanf("%d",n);
   for(i=0;i<*n;i++){       /* FOR EXPECTED # OF PROCESSES          */
      one_periodic(i,&T[0],&C[0]);/* TAKE ONE AT THE TIME         */
   }
}

/*********************************************************************
*  THIS FUNCTION SORTS THE PERIODIC PROCESS IN THE INCREASING ORDER *
*  OF PERIOD. IF IT FINDS THAT TWO PROCESSES ARE NOT IN THE CORRECT *
*  ORDER CALLS THE FUNCTION flip_per TO MAKE THE APPROPRIATE CHANGES.*
*********************************************************************/
void    short_periodics(n,C,T)
int     n;
double  *C,*T;
{
   int        i,j,a,b;
   void       flip_per();
   for(i=n-1;i>0;i--){       /* FOR ALL THE PERIODICS                       */
      for(j=0;j<i;j++){      /* FOR ALL ABOVE THE SELECTED ONE              */
         a=T[j];             /* BOUBLE SORTING                              */
         b=T[j+1];
         if( a > b) flip_per(j,&C[0],&T[0]);
      }
   }
}

/**********************************************************************
*  THIS FUNCTION SORTS THE APERIODIC PROCESS IN THE INCREASING        *
*  ORDER OF LAXITY. IF IT FINDS THAT TWO PROCESSES ARE NOT IN THE     *
*  CORRECT ORDER CALLS THE FUNCTION flip_aper() TO MAKE THE           *
*  APPROPRIATE CHANGES.                                               *
**********************************************************************/
void    short_aperiodics(aper_no,C_ap,T_ap,D_ap)
int     aper_no;
double  *C_ap,*T_ap,*D_ap;
{
   int        i,j,a,b;
   void       flip_aper();

   for(i=aper_no-1;i>0;i--){/* FOR ALL THE APERIODICS                     */
      for(j=0;j<i;j++){      /* FOR ALL ABOVE THE SELECTED ONE              */
         a=D_ap[j]-C_ap[j];
         b=D_ap[j+1]-C_ap[j+1];
         if( a > b) flip_aper(j,&C_ap[0],&T_ap[0],&D_ap[0]);
      }
   }
}

/**********************************************************************
*  THIS FUNCTION CHANGES ALL THE ELEMENTS OF THE ARRAYS THAT CONCERN *
*  PERIODIC TASKS THAT ARE NOT IN THE APPROPRIATE ORDER AT THE TIME  *
*  OF SORTING. SPECIFICLY CHANGES THE PERIOD T THE EXECUTION TIME C  *
*  THE NAMES IN arg1, THE PERIODS AND THE EXECUTION TIMES IN THE     *
*  Tchar AND Cchar RESPECTIVELY.                                      *
**********************************************************************/
void    flip_per(position,C,T)
int     position;
double  *C,*T;
{
   double     temp_T,temp_C;
   char       temp_name[15];
   char       temp[5];

   temp_T=T[position];
   T[position]=T[position+1];
   T[position+1]=temp_T;
   temp_C=C[position];
   C[position]=C[position+1];
   C[position+1]=temp_C;
   strcpy(temp_name,arg1[position]);
   strcpy(arg1[position],arg1[position+1]);
   strcpy(arg1[position+1],temp_name);
   strcpy(temp,Cchar[position]);
   strcpy(Cchar[position],Cchar[position+1]);
   strcpy(Cchar[position+1],temp);
   strcpy(temp,Tchar[position]);
   strcpy(Tchar[position],Tchar[position+1]);
   strcpy(Tchar[position+1],temp);
}

/**********************************************************************
*  THIS FUNCTION CHANGES ALL THE ELEMENTS OF THE ARRAYS THAT CONCERN *
*  APERIODIC TASKS THAT ARE NOT IN THE APPROPRIATE ORDER AT THE TIME *
*  OF SORTING. SPECIFICLY CHANGES THE AVERAGE PERIOD T_ap THE        *
*  EXECUTION TIME C_ap, THE RESPONSE TIME D_ap THE NAMES IN arg2, THE*
*  PERIODS AND THE EXECUTION TIMES IN THE Tchar_ap AND Cchar_ap      *
*  CHARACTER ARRAYS RESPECTIVELY.                                    *
**********************************************************************/
void    flip_aper(position,C,T,D)
int     position;
double  *C,*T,*D;
{
   double     temp1;
   char       temp_name[15];
   char       temp[5];

   temp1=T[position];
   T[position]=T[position+1];
   T[position+1]=temp1;
   temp1=C[position];
   C[position]=C[position+1];
   C[position+1]=temp1;
   temp1=D[position];
   D[position]=D[position+1];
   D[position+1]=temp1;
   strcpy(temp_name,arg2[position]);
   strcpy(arg2[position],arg2[position+1]);
   strcpy(arg2[position+1],temp_name);
   strcpy(temp,Cchar_ap[position]);
   strcpy(Cchar_ap[position],Cchar_ap[position+1]);
   strcpy(Cchar_ap[position+1],temp);
   strcpy(temp,Tchar_ap[position]);
   strcpy(Tchar_ap[position],Tchar_ap[position+1]);
   strcpy(Tchar_ap[position+1],temp);
}
```

```
/************************************************************************
*       THE FOLLOWING FUNCTION RETURNS TO THE PROGRAM THE AVERAGE      *
*   TASK PERIODS T_ap[], THE EXECUTION TIME C_ap[] THE MAX EXECUTION   *
*   DELAY D_ap[] AND THE NAMES OF THE APERIODIC PROCESSES.             *
*   FIRST ASK THE USER FOR THE NUMBER OF APERIODIC PROCESSES THAT THE *
*   SET WILL HAVE AND THEN CALLS THE FUNCTION one_aperiodic            *
*   TO HAVE THOSE VALUES ONE AT THE TIME.                              *
************************************************************************/
void    take_aperiodic_values(n,C,T,D)
int     *n;
double  *C,*T,*D;
{
   int         i;
   void        one_Aperiodic();

   printf("\n\nNumber of aperiodic processes to schedule:");
   scanf("%d",n);
   printf("\n");
   for(i=0;i<*n;i++){       /* FOR EXPECTED # OF APERIODIC PROCESSES*/
      one_Aperiodic(i,&C[0],&T[0],&D[0]);
   }
}

/************************************************************************
*       THE FOLLOWING FUNCTION RETURNS TO THE PROGRAM THE ALL THE      *
*   NAMES AND THE TASKS ATTRIBUTES WHEN THE USER DECIDES TO HAVE THEM *
*   FROM A FILE. FIRST ASK THE USER FOR THE NAME OF THE FILE, THEN     *
*   OPENS THE FILE, READS THE ATTRIBUTES AS CHARACTERS AND THEN        *
*   CONVERTS THEM INTO FLOATING POINT NUMBERS.                         *
************************************************************************/
void    take_from_file(per_no,C,T,aper_no,C_ap,T_ap,D_ap)
int     *per_no,*aper_no;
double  *C,*T,*C_ap,*T_ap,*D_ap;
{
   char        *name[15],*temp[5];
   FILE        *infile;
   int         i;

   printf("\n\t Which file do you want to use ? :");
   scanf("%s",name);
   if(infile = fopen(name,"r")) /* OPEN THE FILE FOR READING          */
   {
      fscanf(infile,"%d\n",per_no); /* FIND # OF PERIODIC PROCESSES */
      for(i=0;i<*per_no;i++){   /* READ ALL THE PERIODIC ATTRIBUTES */
         fscanf(infile,"%s%s%s\n",arg1[i],Cchar[i],Tchar[i]);
         C[i]=atof(Cchar[i]);    /* CONVERT ATTRIBUTES INTO REAL #    */
         T[i]=atof(Tchar[i]);
      }
      fscanf(infile,"%d\n",aper_no);/* FIND # OF APERIODIC PROCESSES*/
      for(i=0;i<*aper_no;i++){ /* READ ALL THE APERIODIC ATTRIBUTES */
         fscanf(infile,"%s%s%s%s\n",arg2[i],Cchar_ap[i],
                                    Tchar_ap[i],temp);
          C_ap[i]=atof(Cchar_ap[i]);/* CONVERT ATTRIBUTES INTO REAL# */
          T_ap[i]=atof(Tchar_ap[i]);
          D_ap[i]=atof(temp);
      }
   }
   fclose(infile);
}

/*********************************************************************
*  THIS FUNCTION PROVIDES THE OPTION TO THE USER TO SAVE THE TASK    *
*  SET NAMES WITH ALL THE ATTRIBUTES, AT THE EXIT, INTO A FILE       *
*  THE USER HAS ONLY TO SPECIFY THE FILE NAME IN WHICH THE SET WILL  *
*  BE SAVED.                                                         *
*********************************************************************/
void    place_to_file(per_no,aper_no,D_ap)
int     *per_no,*aper_no;
double  *D_ap;
{
   char       *name[15],*temp[5];
   FILE       *outfile;
   int        i;

   printf("\n\tFile to save the tasks in => :");
   scanf("%s",name);
   if(outfile = fopen(name,"w"))/* OPEN FILE FOR WRITING             */
   {
      fprintf(outfile,"%d\n",*per_no);/* WRITE # OF PERIODIC TASKS  */
      for(i=0;i<*per_no;i++){ /* WRITE ATTRIBUTES OF PERIODIC TASKS */
         fprintf(outfile,"%s %s %s\n",arg1[i],Cchar[i],Tchar[i]);
      }
      fprintf(outfile,"%d\n",*aper_no);/* WRITE # OF PERIODIC TASKS */
      for(i=0;i<*aper_no;i++){ /*WRITE ATTRIBUTES OF APERIODIC TASKS*/
         fprintf(outfile,"%s %s %s %f\n",arg2[i],
         Cchar_ap[i],Tchar_ap[i],D_ap[i]);
      }
   }
   fclose(outfile);         /* CLOSE FILE                           */
}

/*********************************************************************
*  THIS FUNCTION IS EXECUTED WHEN THE USER SELECTS 2 IN THE MAIN     *
*  MENU. IT HAS INPUTS ALL THE TASK NAMES WITH THEIR ATTRIBUTES.     *
*  A CALL IN THE stop_screen FUNCTION PREVENTS THE OUTPUT FROM       *
*  DISAPPEARING ON THE VIDEO DISPLAY.                                *
*********************************************************************/
void    view_tasks(n,C,T,aper_no,C_ap,T_ap,D_ap)
int     n,aper_no;
double  *C,*T,*C_ap,*T_ap,*D_ap;
{
   int        i;
   void        stop_screen();

   printf("\n\n\n\n\n\n\n\n\n");
   printf("          ----------- PERIODIC PROCESSES ----------\n");
   printf("\t\t   NAME  \t\tPERIOD\t  EXECUTION TIME\n");
   for(i=0;i<n;i++)
      printf("\t\t|%10s|%10.1f\t|%10.1f\t\t|\n",arg1[i],T[i],C[i]);
   printf("          ----------------------------------------\n\n\n");
   printf("    ------------------ APERIODIC  PROCESSES ---------------
                                                          ---\n");
   printf("\t   NAME  \t\tPERIOD\t   EXECUTION TIME \tRESPONSE
                                                         TIME\n");
   for(i=0;i<aper_no;i++)
      printf("\t|%10s|%10.1f\t|%10.1f\t\t  |%10.1f\t |\n",
                               arg2[i],T_ap[i],C_ap[i],D_ap[i]);
   printf("    -------------------------------------------------------
                                                        ---\n\n");
   stop_screen();

}

/*********************************************************************
*  THIS FUNCTION ALLOWS THE USER THE CHANGE THE TASK ATTRIBUTES.      *
*  THE FUNCTION ASKS THE USER TO SPECIFY: IF HE WANTS TO CHANGE       *
*  PERIODIC OR APERIODIC PROCESS, AND THE NUMBER OF THE PROCESS THAT *
*  HE WANTS TO CHANGE. THEN IT CALLS THE APPROPRIATE FUNCTION TO      *
*  HAVE THE NEW ATTRIBUTES. FINALLY MAKES SORTING TO THE PROCESSES.  *
*********************************************************************/
void    change_parameters(per_no,C,T,aper_no,C_ap,T_ap,D_ap)
int     per_no,aper_no;
double  *C,*T,*C_ap,*T_ap,*D_ap;
{
   int         n;
   char        job;

   void        one_periodic();
   void        one_Aperiodic();
   void        short_periodics();
   void        short_aperiodics();

   job         = menu(3);    /* CHANGE PERIODIC OR APERIODIC ??      */
   printf("      WHICH TASK DO YOU WANT TO CHANGE (number)=>:");
   scanf("%d",&n);
   printf("\n");
   switch (job){
      case '1':              /* IF PERIODIC                          */
         if((n<=per_no)&&(n>0)) {
                             /* GET ATTRIBUTES OF ONE PERIODIC TASK  */
            one_periodic(n-1,&T[0],&C[0]);
            short_periodics(per_no,&C[0],&T[0]);
         }
         else printf("   THIS TASK DOES NOT EXISTS \n");
         break;
      case '2':               /* IF APERIODIC                          */
         if((n<=aper_no)&&(n>0)) {
                              /* GET ATTRIBUTES OF ONE APERIODIC TASK */
            one_Aperiodic(n-1,&C_ap[0],&T_ap[0],&D_ap[0]);
            short_aperiodics(aper_no,&C_ap[0],&T_ap[0],&D_ap[0]);
         }
         else printf("   THIS TASK DOES NOT EXISTS \n");
         break;
      default:
         printf("\n      NOT a proper command! \n\n");
   }
   stop_screen();
}

/*********************************************************************
*  THIS FUNCTION TAKES FROM THE KEYBOARD THE NAME AND THE ATTRIBUTES *
*  OF ONE PERIODIC PROCESS AT THE TIME. IT HAS AS INPUTS THE         *
*  POSITION IN  THE ARRAYS AND THE ARRAYS THAT HAS TO FILL.          *
*********************************************************************/
void    one_periodic(i,T,C)
int     i;
double  *C,*T;
{

   printf("\n\n\tEnter the name of the process # %d =>:",(i+1));
   scanf("%s",arg1[i]);
   printf("\n\tEnter the period of process # %d  [ticks] =>:",(i+1));
   scanf("%s",Tchar[i]);
   T[i]=atof(Tchar[i]);
   printf("\n\tEnter the execution time of process #%d [ticks] =>:",
                                                              (i+1));
   scanf("%s",Cchar[i]);
   C[i]=atof(Cchar[i]);
}

/*********************************************************************
*  THIS FUNCTION TAKES FROM THE KEYBOARD THE NAME AND THE ATTRIBUTES *
*  OF ONE APERIODIC PROCESS AT THE TIME. IT HAS AS INPUTS THE        *
*  POSITION IN THE ARRAYS,  AND THE ARRAYS THAT HAS TO FILL.         *
*********************************************************************/
void one_Aperiodic(i,C,T,D)
int i;
double *C,*T,*D;
{
   printf("\n\n\tEnter the name of the process # %d =>:",(i+1));
   scanf("%s",arg2[i]);
   printf("\n\tEnter the average period of process # %d [ticks]=>:",
                                                              (i+1));
   scanf("%s",Tchar_ap[i]);
   T[i]=atof(Tchar_ap[i]);
   printf("\n\tEnter the execution time [ticks] =>:");
   scanf("%s",Cchar_ap[i]);
   C[i]=atof(Cchar_ap[i]);
   printf("\n\tEnter the max responce time of process #%d [ticks]=>:",
                                                                (i+1));
   scanf("%F",&D[i]);
}

/*********************************************************************
*  THIS FUNCTION CALLS ALL THE REQUIRED FUNCTIONS IN ORDER TO MAKE    *
*  SCHEDULABILITY ANALYSIS. IT ALSO CALLS THE FUNCTION THAT RETURNS   *
*  THE PRIORITIES THAT WILL BE ASSIGNED IF THE USER DECIDES TO RUN    *
*  THE SET OF TASKS.                                                  *
*********************************************************************/
void    make_sched_analysis(C,T,n,C_ap,T_ap,D_ap,
                              aper_no,process_pri,tot_T,tot_C)
int     *process_pri,*n,aper_no;
double  *C,*T,*C_ap,*T_ap,*D_ap,*tot_T,*tot_C;
{
   int         i,count,LCM,check;
   double      per_util;
   double      sched_pt[SIZE*2],Lint[SIZE];

   void        create_total_array();
   void        find_Wi();
   void        find_Li();
   void        find_min_Li();
   int         find_if_schedulable();
   int         least_common_mult();
   void        check_aper();
   void        assign_priorities();

                        /* CREATE AN ARRAY IN WHICH BOTH PERIODIC AND
                                APERIODIC PROCESSES WILL BE PLACED */
   create_total_array(n,&C[0],&T[0], &tot_C[0],&tot_T[0]);
          /* CREATE ALL THE INFOS NEEDED FOR THE PERIODIC ANALYSIS */
   count=find_sched_points(&T[0],&sched_pt[0],*n);
   find_Wi(&C[0],&T[0],&sched_pt[0],*n,count);
   find_Li(&sched_pt[0],*n,count);
   find_min_Li(&Lint[0],*n,count);
     /* WITH THOSE INFOS FIND IF PERIODIC PROCESSES ARE SCHEDULABLE */
   check=find_if_schedulable(&per_util,&Lint[0],&C[0],&T[0],*n);
      /* FIND LEAST COMMON MULTIPLIER OF PERIODIC PROCESSES PERIODS */
   LCM=least_common_mult(&T[0],*n);

   /* IF PERIODIC SET SCHEDULABLE MAKE SCHEDULABILITY ANALYSIS FOR THE
    APERIODIC SET AND PLACE THE APERIODIC TASKS IN THE COMMON ARRAYS*/
   if(check == TRUE) check_aper(LCM,per_util,&tot_C[0],&tot_T[0],
                           &C_ap[0],&T_ap[0],&D_ap[0],n,aper_no);
                       /* FIND THE PRIORITIES THAT WILL BE ASSIGNED*/
   assign_priorities(&tot_T[0],&tot_C[0],&process_pri[0],LCM,*n);
}

/*************************************************************************
*  THIS FUNCTION CREATES ARRAYS WITH ELEMENTS ALL THE PERIODIC NAMES *
*  AND ATTRIBUTES. LATER IN THOSE ARRAYS WILL BE PLACED THE          *
*  APERIODIC NAMES AND ATTRIBUTES.                                   *
*************************************************************************/
void    create_total_array(per_no,C,T,tot_C,tot_T)
int     *per_no;
double  *C,*T,*tot_T,*tot_C;
{
   int        i;

   for(i=0;i<*per_no;i++){  /* FOR # OF PERIODIC PROCESSES          */
      tot_T[i]=T[i];
      tot_C[i]=C[i];
      strcpy(name[i],arg1[i]);
      strcpy(tot_Tchar[i],Tchar[i]);
      strcpy(tot_Cchar[i],Cchar[i]);
   }
}

/*************************************************************************
*  THE FUNCTIONS: find_sched_points(), find_Wi(), find_min_Li(),     *
*  find_Li() and find_if_schedulable() ARE NEEDED FOR SCHEDULABILITY *
*  ANALYSIS OF THE PERIODIC PROCESSES. ALL OF THEM ARE PART OF THE   *
*  rate_mono  PROGRAM IN Ref.[Le91]. ONLY MINOR MODIFICATIONS HAVE   *
*  BEEN DONE  AND THEY HAVE BEEN SEPARATED TO FORM MODULAR FUNCTIONS.*
*  FOR THAT REASON THEY HAVE BEEN LEFT WITHOUT COMMENTS.             *
*************************************************************************/

int     find_sched_points(T,sched_pt,n)
double  *T,*sched_pt;
int     n;
{
   int        i,j,k,p,flag,count;
   double     S;

   count=0;
   for(i=0;i<n;i++){
      for(j=0;j<=i;j++){
         for(k=1;k<=floor(T[i]/T[j]);k++){
            flag=0;
            S=k*T[j];
            for(p=1;p<count;p++){
               if(S==sched_pt[p])
               flag=1;
            }
            if(flag==0){
               sched_pt[count]=S;
```

```
                count++;
             }
          }
       }
   }
   return(count);
}

void    find_Wi(C,T,sched_pt,n,count)
double  *C,*T,*sched_pt;
int     n,count;
{
   int        i,j,t;
   double     term1;

   term1=0;
   for(i=0;i<n;i++){
      for(t=0;t<count;t++){
         for(j=0;j<i;j++){
            term1=term1+C[j]*ceil(sched_pt[t]/T[j]);
         }
         W[i][t]=term1;
         term1=0;
      }
   }
}

void    find_Li(sched_pt,n,count)
double  *sched_pt;
int     n,count;
{
   int        i,t;

   for(i=0;i<n;i++){
      for(t=0;t<count;t++){
         Li[i][t]=(W[i][t]/sched_pt[t]);
      }
   }
}


void    find_min_Li(Lint,n,count)
double  *Lint;
int     n,count;
{
   int        i,t;

   for(i=0;i<n;i++) Lint[i]=10000;
   for(i=0;i<n;i++){
      for(t=0;t<count;t++){
         if(Li[i][t]<Lint[i])
```

```
            Lint[i]=Li[i][t];
        }
    }
}

int     find_if_schedulable(per_util,Lint,C,T,n)
double  *Lint,*C,*T,*per_util;
int     n;
{
    int        i,check;
    double     L,U;

    L=0;
    for(i=0;i<n;i++){
       if(Lint[i]>L)
       L=Lint[i];
    }
    U=0;
    for(i=0;i<n;i++) U=U+C[i]/T[i];
    *per_util=U;
    if((L<=1) & (U<=1)){
       printf("\tPeriodic process set SCHEDULABLE.\n\n");
       check=TRUE;
    }
    if((L>1)||(U>1)){
       printf("\tPeriodic Process set NOT schedulable\n\n");
       check=NO;
    }
    printf("\tRemaining processor time for aperiodics : %.2f",
                                                    (1-U)*100);
    printf(" percent\n\n");
    stop_screen();
    return(check);
}


/*********************************************************************
*  THE FOLLOWING FUNCTION COMPUTES THE LEAST COMMON MULTIPLE OF THE  *
*  PERIODIC PROCESSES PERIODS. THE PERIODS ARE INSERTED IN  A        *
*  TEMPORARY ARRAY. BY DIVIDING THE ARRAY ELEMENTS WITH ALL THE      *
*  INTEGERS FROM 1 TILL MAX PERIOD AND BY LOOKING WHEN THIS DIVISION *
*  IS EXACT WE CAN FIND THE LCM.                                     *
*********************************************************************/
int     least_common_mult(T,n)
double  *T;
int     n;
{
    int        i,j,l,flag,temp;
    int        temp_period[SIZE];

    for(l=0;l<n;l++) temp_period[l]=T[l];/* CREATE TEMPORARY ARRAY   */
```

```
   temp=1;                  /* temp = LCM                           */
   for(i=2;i<=T[0];){       /* fOR ALL INTEGERS UNTIL T MAX         */
      flag=0;
      for(j=0;j<n;j++){     /* FOR ALL ELEMENTS FOR THAT INTEGER    */
         if((temp_period[j]/i)*i == temp_period[j]){
                            /* CHECK IF ANY ELEMENT DIVISIBLE       */
            temp_period[j]=temp_period[j]/i;
            flag=1;         /* IF YES DIVIDE AND KEEP THE INTEGER   */
         }
      }
      if (flag == 1) temp=temp*i;/* MULTIMPLY LCM BY THE INTEGER    */
      else i++;
   }
   return(temp);
}

/**********************************************************************
* THIS FUNCTION FINDS IF THE TASK SET TOGETHER WITH THE APERIODIC     *
* PROCESSES IS SCHEDULABLE. FIRST CHECKS THE TOTAL PROCESSOR UTILIZA-*
* TION. IF IS > 1.0 EXITS THE FUNCTION. IF IT IS NOT, CHECK IF THE    *
* APERIODIC TIMING REQUIREMENTS CAN BE SATISFIED. EITHER WAY CONTI-   *
* NUES.  THEN FOR EVERY APERIODIC PROCESS TRIES TO FIND THE OPTIMUM   *
* AMONG THE OTHER PROCESSES POSITION.  ACCORDING TO  THAT THE         *
* PRIORITIES WILL BE ASSIGNED, AND PLACES THE PROCESS THERE.          *
**********************************************************************/
void    check_aper(LCM,per_util,C,T,C_ap,T_ap,D_ap,n,aper_no)
double  per_util,*C,*T,*C_ap,*T_ap,*D_ap;
int     LCM,*n,aper_no;
{
   double      U,min_D,total_laxity;
   int         i,j,a,b,place;
   char        buffer[5];

   int         look_blocking();
   int         find_place_for_aperiodic();
   double      find_total_laxity();
   void        place_aperiodic();

   U=0;
                            /* FIND APERIODIC PROCESSOR UTILIZATION */
   for(i=0;i<aper_no;i++) U=U+C_ap[i]/T_ap[i];
   if((per_util+U)<=1) {    /* IF TOTAL PROCESSOR UTILIZATION < 1.0 */
      printf("\tTotal process set with aperiodics SCHEDULABLE\n\n");
         /* CHECK IF APERIODIC TIMING REQUIREMENTS CAN BE SATISFIED */
      if((place=look_blocking(aper_no,&C_ap[0],&D_ap[0])) != -1){
                            /* IF THEY CANNOT PRINT                 */
         printf("\tBut responce time of %s process",arg2[place]);
         printf(" may NOT be acheived\n\n");
      }
      for(i=0;i<aper_no;i++){/* FOR ALL THE APERIODIC PROCESSES     */
                            /* FIND LAXITY OF COMBINED PROCESS      */
      total_laxity=find_total_laxity(i,aper_no,&C_ap[0],&D_ap[0]);
      for(j=i;j<aper_no;j++){/* find min resp. time for conbined */
         a=D_ap[j];
         b=min_D;
         if( a<b ) min_D=D_ap[j];
      }        /* FIND PLACE IN THE ARRAY FOR COMBINED APERIODIC */
      place=find_place_for_aperiodic(min_D,&C[0],&T[0],
                                         total_laxity,*n,place+1);
                           /* PLACE APERIODIC IN THAT POSITION     */
      place_aperiodic(*n,place,i,&C[0],&T[0],&C_ap[0],&T_ap[0]);
      *n=*n+1;  /* LOWEST POSITION NEXT  APERIODIC CAN BE PLACED */
    }
                           /* THIS IS FOR USER INFORMATION         */
    printf("\tDo you want to see the whole set (y,n)? \n\n");
    readln(0,buffer,2);
    if((buffer[0]=='y')||buffer[0]=='Y'){
      printf("\t\t    NAME  \t\tPERIOD\t  EXECUTION TIME\n");
      for(i=0;i<*n;i++) printf("\t\t|%10s|%10.1f\t|%10.1f\t\t|\n",
                                             name[i],T[i],C[i]);
    }
  }
  else{                    /* IF TOTAL PROCESSOR UTILIZATION > 1.0 */
    printf("\tTotal set NOT schedulable\n\n");
    printf("\tThe set exceeds processor time by : %.2f percent\n\n",
                                            (U+per_util-1)*100);
  }
  stop_screen();
}

/**********************************************************************
*  THIS FUNCTION COMPUTES THE TOTAL LAXITY (MIN. RESPONSE TIME - SUM *
*  OF EXECUTIONS) OF THE COMBINED APERIODIC PROCESS                   *
**********************************************************************/
double  find_total_laxity(base,aper_no,C_ap,D_ap)
int     base,aper_no;
double  *C_ap,*D_ap;
{
  int        j,a,b;
  double     min_D,sum_C;

  sum_C=0.0;
  min_D=100000.0;          /* JUST A BIG NUMBER                     */
  for(j=base;j<aper_no;j++){/* FOR APERIODICS THAT HAVEN'T CHECKED */
    a=D_ap[j];
    b=min_D;               /* FIND MINIMUM RESPONSE TIME            */
    if( a<b ) min_D=D_ap[j];
    sum_C=sum_C+C_ap[j];  /* COMPUTE SUM OF EXECUTION TIMES       */
  }                        /* RETURN TOTAL LAXITY IF > 0            */
  if((a=(min_D-sum_C))>0) return(min_D-sum_C);
  else return(0.0);        /* RETURN 0                              */
}
```

```
/***********************************************************************
*  THIS FUNCTION FINDS IF THE APERIODIC SET CAN MEET ITS TIMING        *
*  REQUIREMENTS. IF THE SUM OF THE EXECUTION TIMES OF THE PROCESSES    *
*  WITH HIGHER TIMING REQUIREMENTS IS GREATER THAN THE RESPONSE TIME   *
*  OF A PROCESS i THEN THE PROCESS i MAY NOT MEET ITS RESPONSE TIME    *
*  WHEN ALL START SIMULTANEOUSLY. IN THAT CASE THE PROGRAM RETURN      *
*  THE NUMBER OF THE i PROCESS. OTHERWISE RETURNS -1                   *
***********************************************************************/
int     look_blocking(aper_no,C_ap,D_ap)
int     aper_no;
double  *C_ap,*D_ap;
{
   int        i,j,a;
   double     sum_C;

   for(i=aper_no-1;i>0;i--){/* FOR ALL THE APERIODIC PROCESSES       */
      sum_C=0.0;
                             /* FIND SUM OF THE EXECUTIONS TIMES FOR
                                         PROCESSES WITH LOWER LAXITY */
      for(j=0;j<i;j++) sum_C=sum_C+C_ap[j];
      a=D_ap[i]-C_ap[i]-sum_C;
      if(a<0) return(i);     /* IF RESPONSE TIME SMALLER THAN THE SUM
                                    RETURN THE NUMBER OF THE PROCESS */
   }
   return(-1);               /* ELSE RETURN -1                       */
}


/***********************************************************************
*  THIS PROCESS COMPUTES THE PRIORITIES THAT WILL ASSIGNED TO THE      *
*  PROCESSES IF THE USER DECIDES TO RUN THE SET. THE CODE IS FROM      *
*  rate_mono PROGRAM in Ref.[Le91] WITH THE MODIFICATION THAT IT HAS   *
*  BECOME A FUNCTION AND INSTEAD OF GIVING AS INPUT ONLY THE PERIODIC*
*  PROCESSES, THE COMBINED ARRAY OF PROCESSES IS PROVIDES. FOR THAT    *
*  REASON THE COMMENTS HAVE BEEN OMITTED.                              *
***********************************************************************/
void    assign_priorities(T,C,process_pri,LCM,n)
double  *T,*C;
int     *process_pri;
int     LCM,n;
{
   double     temp,temp1,temp2;
   int        i,j,k,l;

   process_pri[0]=MAX_PRIORITY;
   temp=1;
   temp1=0;
   temp2=1;
   for(i=1;i<n;i++){
      for(j=0;j<=(i-1);j++) temp2=temp2*T[j];
      for(k=0;k<=(i-1);k++){
         for(l=0;l<=(i-1);l++){
         if(l!=k) temp=temp*T[l];
      }
      temp1=temp1+C[k]*temp;
      temp=1;
   }
   process_pri[i]=process_pri[0]-ceil(LCM*(temp1/temp2)*5*i);
   temp1=0;
   temp2=1;
   }
   temp1=0;
   for(i=0;i<n;i++){
      if(T[i]>temp1)
      temp1=T[i];
   }
   for(i=1;i<n;i++) process_pri[i]=process_pri[0]-(i*temp1);
}


/**********************************************************************
*  THIS FUNCTION INSERT THE NAME AND THE ATTRIBUTES OF AN APERIODIC  *
*  PROCESS IN THE ARRAYS THAT CONTAINS BOTH PERIODIC AND APERIODIC   *
*  PROCESSES. THIS IS DONE BY MOVING ALL THE ELEMENTS THAT ARE BELOW *
*  THE SPECIFIED POSITION ONE STEP DOWN  AND THEN INSERTING THE NEW  *
*  ELEMENTS.                                                         *
**********************************************************************/
void    place_aperiodic(n,place,aper,C,T,C_ap,T_ap)
double  *C,*T,*C_ap,*T_ap;
int     n,place,aper;
{
   int        i;

   for(i=n;i>place;i--){/* MOVE ELEMENTS BELOW place ONE STEP DOWN  */
      strcpy(name[i],name[i-1]);
      C[i]=C[i-1];
      T[i]=T[i-1];
      strcpy(tot_Cchar[i],tot_Cchar[i-1]);
      strcpy(tot_Tchar[i],tot_Tchar[i-1]);
   }                         /* INSERT NEW ELEMENTS                     */
   strcpy(name[place],arg2[aper]);
   strcpy(tot_Cchar[place],Cchar_ap[aper]);
   strcpy(tot_Tchar[place],Tchar_ap[aper]);
   C[place]=C_ap[aper];
   T[place]=T_ap[aper];
}
```

```
/**********************************************************************
*  THIS FUNCTION FINDS THE BEST POSITION FOR A  COMBINED  APERIODIC  *
*  PROCESS TO HAVE ITS PRIORITY. STARTS LOOKING IF THE RESPONSE TIME *
*  CAN BE ACHIEVED BY GIVING THE LOWEST PRIORITY. IF NOT IS GOING    *
*  POSITION LOWER.  IT CANNOT GO LOWER THAN THE LAST ASSIGNED PRIO-  *
*  RITY. THAT IS FOR THE CASE THE APERIODIC PROCESSES CANNOT         *
*  ACCOMPLISH THE REQUIRED RESPONSE TIME BUT THE USER STILL WANTS TO *
*  RUN THE SET.                                                      *
**********************************************************************/

int     find_place_for_aperiodic(minD,C,T,Delay,n,last)
double  *C,*T,minD,Delay;
int     n,last;
{
   int        i;
   double     U;

   U=0;
      /* FIND WORST CASE PROCESSOR UTILIZATION DURING RESPONSE TIME*/
   for(i=0;i<n;i++) U=U+ceil(minD/T[i])*C[i];
   while((Delay<(U)) && (n>last)){ /* WHILE TIME NEEDED FOR OTHER
                              PROCESSES GREATER THAN RESPONSE TIME */
      n--;                  /* GO ONE POSITION LOWER               */
      U=0;      /* FIND PROCESSOR UTILIZATION WITH ONE PROCESS LESS */
      for(i=0;i<n;i++) U=U+ceil(minD/T[i])*C[i];
   }
   return(n);               /* RETURN THE OPTIMUM POSITION         */
}

/**********************************************************************
*  THIS FUNCTION FIRST CALLS THE TASK THAT CREATE ALL THE PIPES.     *
*  THEN SAVES SOME STATISTISTICAL  INFORMATION IN A FILE FOR LATER   *
*  USER ANALYSIS. THEN CREATES ALL THE TASKS.                        *
*  THEN SENDS SIGNALS TO ALL OF THEM TO START EXECUTION FINALLY      *
*  WAITS FOR PROCESSES TO FINISH EXECUTION AND CLOSES THE PIPES.     *
**********************************************************************/
void    run_set(C,T,process_pri,parent,n)
int     *process_pri,n;
double  *C,*T;
procid  *parent;
{
   FILE       *file;
   int        pid,pipes_id,prid[SIZE];

   int        create_pipes();
   void       save_priorities();
   void       fork_the_processes();
   void       start_processes();
   void       receive_send_kill();

   pid=getpid();            /* GET PROCESS ID                      */
   pipes_id=create_pipes(); /* CREATE ALL THE PIPES                  */
   if( file=fopen("statist", "w"))/* SAVE IN FILE ASSIGNED PRIORIT.*/
      save_priorities(file,&C[0],&T[0],&process_pri[0],n);
   else
      printf("\tcannot open %s to save priority info\n\n","statist");
                            /* CREATE ALL THE TASKS                  */
   fork_the_processes(&process_pri[0],&prid[0],parent,n);
   fclose(file);
   start_processes(&prid[0],n);/* START EXECUTION OF TASKS          */
   receive_send_kill(pipes_id,*parent,n);
                    /* WAIT FOR TASK TO END AND CLOSE ALL THE PIPES */
}

/**********************************************************************
*  THIS FUNCTION IT JUST SAVES THE TASKS PRIORITIES AND SOME OF THE  *
*  TASKS ATTRIBUTES IN A FILE FOR LATER ANALYSIS.                    *
**********************************************************************/
void    save_priorities(file,C,T,process_pri,n)
FILE    *file;
double  *C,*T;
int     *process_pri,n;
{
   int         i;

   fprintf(file,
            "process# \t  name  \texec. time\t period \tpriority\n");
   for(i=0;i<n;i++){
      fprintf(file,
            "%6d  \t%5s  \t%9.1f \t %5.1f \t%  %8d\n",(i+1),name[i],
                                        C[i],T[i],process_pri[i]);
   }
   fprintf(file,"\n\n");
}

/**********************************************************************
*  THIS FUNCTION FORKS THE PROCESS pipes.c THAT CREATES ALL THE PIPES*
*  AND THEN IT FORKS THE FUNCTION start.c THAT GIVES INITIAL VALUES  *
*  IN SOME OF THE PIPES SO THE AUV-II CAN START EXECUTION            *
**********************************************************************/
int     create_pipes()
{
   int         pipes_id,start_id;

   if((pipes_id=os9exec(os9forkc,arg4[0],arg4,environ,0,61000,15))<0)
   {                        /* FORK pipes.c TASK                     */
      fprintf(stderr,"Cannot fork %s \n","pipes");
      exit(errno);
   }
   pause();                 /* WAIT FOR pipes.c TO FINISH EXECUTION */
   if((start_id=os9exec(os9forkc,arg3[0],arg3,environ,0,61000,15))<0)
   {                        /* FORK start.c TASK                     */
```

```
      fprintf(stderr,"Cannot fork %s \n","start");
      exit(errno);
   }
   return(pipes_id);
}

/*********************************************************************
*  THIS FUNCTION FORKS ALL THE PROCESSES IN THE DATA-FLOW DIAGRAM.   *
*  THE ARGUMENTS THAT ARE PASSED ON THOSES PROCESSES ARE THEIR NAMES,*
*  THEIR EXECUTION TIMES C, AND THEIR PERIODS T. THE PRIORITIES ARE  *
*  THOSE THAT HAVE ALREADY BEEN COMPUTED.  AFTER EACH FORKING THE    *
*  PROGRAM SUSPENDS ITSELF UNTIL THE PROCESS FINISHES INITIALIZATION.*
*********************************************************************/
void    fork_the_processes(process_pri,prid,parent,n)
int     *process_pri,*prid,n;
procid  *parent;
{
   int        i,pid,token_id;
   char       *lala1[4];

   lala1[0]=(char *)malloc(15);
   lala1[1]=(char *)malloc(5);
   lala1[2]=(char *)malloc(5);
   lala1[3]=NULL;
   pid=getpid();
   for(i=0;i<n;i++){          /* FOR ALL THE PROCESSES                  */
      strcpy(lala1[0],name[i]);/* COPY THE ARGUMENTS IN TEMP. ARRAY */
      strcpy(lala1[1],tot_Cchar[i]);
      strcpy(lala1[2],tot_Tchar[i]);
                              /* FORK THE PROCESSES                     */
      if((prid[i]=os9exec(os9forkc,lala1[0],lala1,environ,0,
                                              process_pri[i],15))<0)
      {
         fprintf(stderr,"Cannot fork %s \n",arg1[i]);
         exit(errno);
      }
   pause();                   /* WAIT FOR PROCESS TO BE INITIALIZED    */
   }
}

/*********************************************************************
*  THIS FUNCTION IS WAITING FOR ALL THE PROCESSES IN THE DATA-FLOW   *
*  DIAGRAM TO FINISH EXECUTION. THEN IT SENDS A SIGNAL IN THE PROCESS*
*  pipes.c THAT CREATED ALL THE PIPES TO FINISH EXECUTION.           *
*********************************************************************/
void    receive_send_kill(pipes_id,parent,n)
int     pipes_id,n;
procid  parent;
{
   intkillerr,i;
   for(i=0;i<n;i++) pause();/* WAIT ALL THE PROCESS TO FINISH        */
   if((killerr=kill(pipes_id,SIGWAKE)) == -1) fprintf(stderr,
      cannot wakeup pipes.c in %s, error No =%d\n", "sched.c",errno);
}                            /* SIGNAL pipes.c TO FINISH EXECUTION    */

/**********************************************************************
*  AFTER THE INITIALIZATION EACH PROCESS IN THE DATA-FLOW DIAGRAM     *
*  SUSPEND ITSELF. THIS FUNCTION SENDS KILL SIGNALS IN ALL THE        *
*  PROCESSES SO THEY CAN START SIMULTANEOUSLY THE EXECUTION OF THEIR *
*  MAIN LOOPS.                                                        *
**********************************************************************/
void    start_processes(prid,n)
int     *prid,n;
{
   int        killerr,i;

   for(i=0;i<n;i++){
      if((killerr=kill(prid[i],SIGWAKE)) == -1) fprintf(stderr,
                 "Cannot wakeup process # %d in %s, error No =%d\n",
                                         prid[i],"sched.c",errno);
   }
}

/**********************************************************************
*  THIS FUNCTION IS BEING CALLED WHEN THE USER DECIDES TO EXIT THE   *
*  SCHEDULER. FIRST PROVIDES THE OPTION TO THE USER TO SAVE THE TASKS*
*  SET. THEN RELEASES ALL THE MEMORY THAT HAS BEEN USED WITH malloc. *
*  THEN IT SENDS A SIGNAL TO THE PARENT PROCESS (newss.c) TO FINISH  *
*  EXECUTION AND THEN IT EXITS.                                       *
**********************************************************************/
void    my_end(n,aper_no,D_ap,parent)
int     n,aper_no;
double  *D_ap;
procid  parent;
{
   char       bufer[5];

   void       place_to_file();
   void       return_memory();
   void       parent_kill();

   printf("\tDo you want to save the tasks in file (y,n) ?\n");
   readln(0,bufer,2);        /* OPTION IN USER TO SAVE THE SET       */
   printf("\n");
   if((bufer[0]=='y')||(bufer[0]=='Y'))
                             place_to_file(&n,&aper_no,&D_ap[0]);
   return_memory();          /* RELEASE THE USED WITH malloc MEMORY  */
   parent_kill(parent);      /* KILL THE START SCHEDULER (newss.c)   */
   exit();
}
```

```
/***********************************************************************
*  THIS FUNCTION SENDS A SIGNAL TO newss.c TO FINISH EXECUTION        *
***********************************************************************/
void    parent_kill(parent)
procid  parent;
{
   int        killerr;

   if((killerr=kill(parent._pid,SIGWAKE)) == -1) fprintf(stderr,
      "Cannot wakeup newss.c in %s, error No =%d\n", "sched.c",errno);
}

/***********************************************************************
*  THIS FUNCTION IS BEING CALLED AFTER EVERY SCHEDULER OUTPUT SO IT  *
*  WILL STOP THE SCROLLING OF THE SCREEN.                             *
***********************************************************************/
void    stop_screen()
{
   char       hit[10];

   printf("\n\t Hit RETURN to continue.\n ");
   readln(0,hit,1);
   printf("\n\n");
}

/***********************************************************************
*  THIS FUNCTION RELEASES ALL THE MEMORY ALLOCATIONS BEFORE EXIT      *
***********************************************************************/
void    return_memory()
{
   int        i;
   for(i=0;i<SIZE;i++){      /* FOR MAX EXPECTED # OF PROCESSES      */
      free(arg1[i]);
      free(arg2[i]);
      free(name[i]);
      free(Cchar[i]);
      free(Cchar_ap[i]);
      free(tot_Cchar[i]);
      free(Tchar[i]);
      free(Tchar_ap[i]);
      free(tot_Tchar[i]);
   }
}
```

APPENDIX C: CODE FOR CALLED PROCESSES

```
/***********************************************************************
*   Program    :  GUIDANCE.C                                           *
*   Purpose    :  CODE FOR CALLED PROCESSES.                           *
*   Author     :  LTJG D. MAKRIS  H.N.                                 *
*   Description:  THIS IS GENERIC DUMMY PROCESS BEING USED TO VERIFY   *
*                 THE RESULTS OF THE SCHEDULER. THIS PROCESS HAS AS    *
*                 INPUT THE PERIOD OF THE PROCESS THIS PROSSES USES    *
*                 THE FUNCTION "get_timing_info" WHICH RETURNS THE     *
*                 TIME THAT THE FUNCTION WAS CALLED. THE OUTPUT OF     *
*                 THIS PROCESS IS IN THE FILE "out3" WHERE TIMIMG      *
*                 INFORMATION IS WRITEN.                               *
*                 THE TIMING PROCESS DIFFERS ONLY IN THE DELAY. IT     *
*                 HAS A TIMING SLEEP EQUAL TO THE PERIOD.              *
*                 THE INITIALIZATION PROCESS IS ALSO SIMILAR. THE      *
*                 DIFERENCE IS THAT THERE IS NOT MAIN LOOP.            *
***********************************************************************/

#include        <stdio.h>
#include        <errno.h>
#include        <modes.h>
#include        <signal.h>
#include        <procid.h>

#define         NUMBER          4
#define         STATISTICS      "out3" /* OUTPUT FILE                  */
#define         STAT_INFO_DELAY 394    /* DELAY OF THE PROGRAM         */
#define         LOOP            3160   /* TIME OF ONE TICK             */
#define         TRUE            1
#define         RUN             50     /* EXPECTED TIMES TO RUN        */

char    *names[]= {"/PIPE/current_posture","/PIPE/emergency_posture",
            "/PIPE/reference_postures","/PIPE/commanded_postures",0,};

extern  int    exit(),pause(),getpid(),_get_process_desc(),kill();

char    *outbuffer[10]={"3a","3b","3c","3d","3e",
                        "3f","3g","3h","3i","3j"};

icpthand(signum)
int     signum;
{
                                /* INTERCEPT  HANDLER                  */
   fprintf(stderr,"I received in guidance.c signal %d\n",signum);
}
```

```
main(argc,argv)
int     argc;
char    *argv[];
{
   int         pipes[NUMBER];
   FILE        *statfile;
   char        buffer[10];
   int         delay,i,j,count,killerr,pdata;
               period,readerr,WriteLength;
   int         time_start[RUN],time_stop[RUN],
               tick_start[RUN],tick_stop[RUN];
   short int   parentid;
   unsigned    final_sleep;

   void        get_timing_info();
   short int   get_parent_id();

   intercept(icpthand);
   i=j=0;
   count       = 0;          /*OPEN THE FILE FOR THE RESULTS          */
   statfile    = fopen(STATISTICS,"a");

   while(names[i]!=NULL){
      if((pipes[i] = open(names[i],S_IREAD+S_IWRITE)) == -1){
         fprintf(stderr,
                 "\tThe %s is not opening in %s for writing \n",
                 names[i],argv[0]);
         exit(errno);
      }                      /*OPEN THE PIPE FOR WRITING              */
      i++;
   }
   delay       = LOOP*atoi(argv[1])-STAT_INFO_DELAY;
         /*CALCULATE THE DELAY ACCORDING TO THE EXEC. TIME IN TICKS*/
   parentid = get_parent_id(argv[0]);
                             /* WAKE UP sched.c                       */
   if((killerr=kill(parentid,SIGWAKE)) == -1)
      fprintf(stderr,"Cannot wakeup rm.c in %s, error No =%d\n",
                                                       argv[0],errno);
   pause();               /* WAIT FOR ALL PROCESSES TO START TOGETHER */

   while(j<RUN-1)
   {
      get_timing_info(&time_start[j],&tick_start[j]);
                             /* READ PIPE FROM PERIODIC PROCESS       */
      if((readerr = readln(pipes[0],buffer,10)) == -1)
         fprintf(stderr,"cannot read %s,in %s, error number %d\n",
                                              names[0],argv[0],errno);
                             /* READ PIPE FROM APERIODIC PROCESS      */
      if(( pdata = _gs_rdy(pipes[1])) > 1){
         if((readerr = readln(pipes[1],buffer,10)) == -1)
```

```
            fprintf(stderr,"cannot read %s,in %s, error number %d\n",
                                              names[1],argv[0],errno);
      }
                               /* READ PIPE FROM PERIODIC PROCESS      */
      if((readerr = readln(pipes[2],buffer,10)) == -1)
         fprintf(stderr,"cannot read %s,in %s, error number %d\n",
                                              names[2],argv[0],errno);
      for(i=1;i<delay;i++); /* INSTEAD OF THE REAL PROGRAM          */

                               /* WRITE IN PIPE                        */
      if((WriteLength = writeln(pipes[3],outbuffer[count%10],10)) < 0)
         fprintf(stderr,"cannot write in %s,in %s, error No %d\n",
                                              names[3],argv[0],errno);

      get_timing_info(&time_stop[j],&tick_stop[j]);
      j++;
   }
   for(j=0;j<RUN-1;j++){     /* THIS IS ONLY FOR EXPERIMENT PURPOSES */
      fseek(statfile,0,2);  /* FIND THE END OF THE out3 FILE         */
      fprintf(statfile,
                 "%s for\t %d time, time:%6d-%6d, ticks %6d-%6d \n",
                 argv[0],j,time_start[j],time_stop[j],tick_start[j],
                                                       tick_stop[j]);
   }                           /* SAVE THE TIMING INFORMATION          */
                               /* SIGNAL sched.c THAT FINISH EXECUTION */
   if((killerr=kill(parentid,SIGWAKE)) == -1)
      fprintf(stderr,"Cannot wakeup rm.c in %s, error No =%d\n",
                                                       argv[0],errno);
   exit(0);
}

/*********************************************************************
*  THIS FUNCTION ACCESSES THE SYSTEM CLOCK AND RETURNS THE TIME AND  *
*  THE TICK.                                                         *
*********************************************************************/
void    get_timing_info(ret_time,ret_tick)
int     *ret_time,*ret_tick;
{
   int         date,time,tick,mask;
   short       day;
   mask        =0x0000ffff;

   _sysdate(3,&time,&date,&day,&tick);
   *ret_tick=(tick & mask);
   *ret_time=time;
}
```

```
/**********************************************************************
*  THIS FUNCTION RETURNS THE PROCESS I.D. OF THE sched.c. THE        *
*  ARGUMENT NAME IS ONLY FOR ERROR HANDLING.                          *
**********************************************************************/
short int     get_parent_id(name)
char    *name;
{
   int         gpiderr;
   short int  pid;
   procid      parent;

   if((pid = getpid()) == -1)/* TAKE THE PROCESS I.D.               */
      fprintf(stderr,"Cannot get Process ID in %s, error No= %d\n",
                                                        name,errno);
                              /* TAKE THE PARENTS (sched.c) I.D.      */
   if((gpiderr = _get_process_desc(pid,sizeof(parent),&parent)) == -1)
      fprintf(stderr,
              "Cannot get Process Descriptor in %s, error No= %d\n",
                                                        name,errno);
   return(parent._pid);      /* RETURN THE PARENTS I.D.              */
}
```

APPENDIX D: CODE CREATING THE PIPES

```
/*********************************************************************
*   Program    :  PIPES.C                                            *
*   Purpose    :  MAIN CODE PROCESS THAT CREATES PIPES.              *
*   Author     :  LTJG D. MAKRIS  H.N.                               *
*   Description:  THIS PROGRAM CREATES ALL THE PIPES THAT WILL BE    *
*                 USED FOR THE PROCESS COMMUNICATION. AFTER THE      *
*                 CREATION THE PROGRAMM REMAINS IDLE UNTIL THE SET   *
*                 FINISHES. OTHERWISE PIPES WILL CLOSE.              *
*********************************************************************/

#include       <stdio.h>
#include       <errno.h>
#include       <modes.h>
#include       <signal.h>
#include       <procid.h>

#define        MAXSIZE   20
#define        NUMBER    16

                            /* DEFINE ALL THE PIPES                  */
char    *names[]= {"/PIPE/token_pipe","/PIPE/commanded_postures",
                   "/PIPE/signals","/PIPE/positions",
                   "/PIPE/current_posture","/PIPE/emergency_posture",
                   "/PIPE/inertial_data","/PIPE/obstacle_alert",
                   "/PIPE/alert","/PIPE/path",
                   "/PIPE/range_data","/PIPE/reference_postures",
                   "/PIPE/replan_request","/PIPE/sonar_data",
                   "/PIPE/status","/PIPE/systems_status",0,};

extern  int     exit(),pause(),getpid(),_get_process_desc(),
                kill(),intercept();

icpthand(signum)
int signum;
{
                            /* INTERCEPT HANDLER                     */
   fprintf(stderr,"I received signal in pipes.c : %d\n",signum);
}

main(argc,argv)
int     argc;
char    *argv[];
{
   FILE       *pipes[NUMBER];
   int        killerr,i;
   short int  parentid;
   short int  get_parent_id();

   intercept(icpthand);
   i=0;
   while(names[i]!=NULL){   /* OPEN ALL THE PIPES LIKE FILES         */
      if((pipes[i] = fopen(names[i],"w")) == NULL){
         fprintf(stderr,"Cannot open %s.Error %d \n\n ",
                                                      names[i],errno);
         exit(errno);
      }
      i++;
   }
   parentid = get_parent_id(argv[0]);/* FIND PARENTS I.D.
                                  CODE HAS DEFINED IN APPEMDIX C */
                    /* SIGNAL sched.c THAT INITIALIZATION FINISHED */
   if((killerr = kill(parentid,SIGWAKE)) == -1)
      fprintf(stderr,"Cannot wakeup rm.c in %s, error No =%d\n",
                                                       argv[0],errno);

   pause();                 /* WAIT OTHERWISE PIPES WILL CLOSE      */
   exit(0);
}
```

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Shridhar B. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	2
5.	Professor Roberto Cristi, Code EC/Cx Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor Anthony Healey, Code ME/Hy Mechanical Engineering Department Naval Postgraduate School Monterey, California 93943-5000	1
7.	Professor Robert B. McGhee, Code CS/Mz Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	1

8.	Embassy of Greece Naval Attache 2228, Massachusetts Av., N.W. Washington, D.C. 20008	2
9.	LTJG Makris Dionysios Tepeleniou 20, Papagos Athens, 15669 GREECE	3